Diplomarbeit

On Polymorphic Types with Enforceable Linearity for a Quantum Lambda Calculus

Marco Voigt

Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl für Automatentheorie

Betreuerin:

Dr.-Ing. Monika Sturm

Verantwortlicher Hochschullehrer:

Prof. Dr.-Ing. Franz Baader

Eingereicht am 05. April 2012
Verteidigt am 15. Mai 2012
(Korrigierte Version vom 31. Mai 2012)

Technische Universität Dresden
Institut für Theoretische Informatik
Lehrstuhl für Automatentheorie

01.11.2011

Aufgabenstellung Diplomarbeit

On Polymorphic Types with Enforceable Linearity for a Quantum Lambda Calculus

Bearbeiter: Marco Voigt
(Studiengang Informatik (PO 2004, StO 2004))

Quantum Computing stellt neben dem DNA Computing eine der wichtigsten Ausprägungen im Natural Computing dar. Die Forschungsarbeiten auf dem Gebiet zum Quantum Computing sind interdisziplinär und unmittelbar verbunden mit Aspekten der Quantenphysik bzw. Quantenmechanik. Aufbauend auf der Belegarbeit *„The Quantum Lambda Calculus theory and application“* (Marco Voigt, TU Dresden, 2011) ist ein polymorphes Typsystem für einen Lambda-Kalkül zum Quantum Computing zu entwickeln, d.h. nach dem Vorbild des Systems F in der Familie der getypten Lambda-Kalküle ist ein System F für den getypten Quantum Lambda Calculus zu definieren.
In der Diplomarbeit sind dabei folgende Aspekte, Konzepte und Eigenschaften einzuführen bzw. zu beachten:

- Vorstellung des Modells Quantum Computing
- Vorstellung des Quantum Lambda Calculus
- Einführung eines Typsystems mit linearen und nichtlinearen Typen
- Berücksichtigung des *Subtyping*-Konzeptes im Typsystem
- Definition polymorpher Typkonstrukte
- Definition und Beweis von Eigenschaften und Aussagen des polymorphen Typsystems
- Bewertung der Ergebnisse

Die Bearbeitung des Themas umfasst das Studium folgender Literaturstellen:

- Pierce, B.C. *Types and Programming Languages.* The MIT Press, Cambridge, London. 2002
- Selinger, P.; Valiron, B. *Quantum lambda calculus.* In: Gay, S.; Mackie, I. (eds.) *Semantic Techniques in Quantum Computation.* Cambridge University Press, Cambridge, New York. (2009), pp. 135–172
- Gruska, J. *Quantum Computing.* McGraw Hill, London. 1999

Organisation und Termine

Betreuerin: Dr. M. Sturm
Verantw. Hochschullehrer: Prof. F. Baader
Bearbeitungszeitraum : 01.11.2011 - 30.04.2012
Vorstellung der Arbeitsergebnisse: 03/2012

Ich versichere, dass ich die vorliegende Diplomarbeit selbstständig erarbeitet und dabei ausschließlich die angegebenen Quellen und Hilfsmittel genutzt habe. Inhaltlich und wörtlich übernommene Ergebnisse und Gedanken Dritter wurden ausdrücklich als solche gekennzeichnet. Die Arbeit wurde bisher weder in Gänze noch in substantiellen Teilen veröffentlicht oder zur Erlangung eines akademischen Grades an einer Hochschule im In- oder Ausland vorgelegt.

Dresden, den 05. April 2012

_____

Marco Voigt

Contents

Notational conventions

Unless stated otherwise in the respective places, we use the following notations.

General notation

m, n	natural numbers ≥ 0
i, j, k, l	integers, mainly used as indices
$\mathcal{P}(S)$	power set of a set S

Quantum computation

α, β	complex scalars
$\left\|\varphi\right\rangle, \left\|\psi\right\rangle$	arbitrary quantum states
$\left\|b\right\rangle$	quantum basis states
$\mathcal{H}$	Hilbert space (over the field $\mathbb{C}$ of complex numbers)
$\mathfrak{B}$	standard basis of a Hilbert space
$\mathcal{H}_n$	2^n dimensional Hilbert space (over $\mathbb{C}$)
$\mathfrak{B}_n$	standard basis of a Hilbert space $\mathcal{H}_n$
U, V	arbitrary unitary operators
Id_n	identity operator over $\mathcal{H}_n$
H	Hadamard operator (over $\mathcal{H}_1$)
$\mathbb{0}$	unique zero vector of a Hilbert space $\mathcal{H}$
$\mathbb{1}$	positive unit vector of a Hilbert space $\mathcal{H}_0$

***Quantum Lambda Calculus* (*QLC*) and typing**

Θ, Ξ	type contexts
Γ, Δ	term contexts
$\Phi, \Psi, \Upsilon, \Sigma$	type terms
s, t	function terms
v, w	value terms
C	type constants
c	term constants
X, Y, Z	type variables
x, y, z	term variables
f	term variables used as placeholders for functions
q	term variables associated to quantum data
Q	quantum states embedded in quantum closures
L	linking sequences, linking free term variables in QLC function terms to qubits in associated quantum data
$\ulcorner n \urcorner$	natural number n encoded as Church numeral

Words over arbitrary alphabets A

A^*	Kleene closure of alphabet A
ε	the empty word
$\left\|w\right\|$	length of word $w \in A^*$
$\left\|w\right\|_a$	number of occurrences of letter $a \in A$ in word $w \in A^*$

Permutations and transpositions

π, σ	permutations
τ	transpositions

Anything that can go wrong, will.
– attributed to Edward A. Murphy, Jr.

Well-typed programs cannot "go wrong."
– Robin Milner (1978)

1 Introduction and motivation

Ever since the dawn of quantum computation in the 1980's its underlying principles have posed new challenges but also offered new prospects to a broad variety of areas in computer science, which are otherwise already well understood. This has not at all changed until today. While general-purpose programming languages designed for classical computation devices and equipped with sophisticated type systems gain more and more maturity, the field of quantum programming languages (typed and untyped) is still in its infancy. There is thus a need for further development especially focusing on the peculiarities quantum physics introduces into the realm of information processing.

In this work we therefore approach polymorphic typing in the presence of restrictions on the duplicability of certain data. We in particular motivate these restrictions with the physical laws on which quantum computation is grounded. However, in general, they could also arise in less "exotic" situations.[1] Our considerations are mainly focused on a functional calculus developed by Benoît Valiron and Peter Selinger in the years 2004 to 2009 as one approach to quantum programming languages. This *Quantum Lambda Calculus*[2] models classical computations in the tradition of Church's λ-calculus as well as quantum computations following the "classical control, quantum data" paradigm. As already indicated, this approach does by far not stand alone since there have been proposed several alternative calculi aiming in the same direction. We elaborate a bit on related work below and give relevant bibliographic references there.

In our work we start from an untyped variant of the *Quantum Lambda Calculus* and seek to develop an appropriate type system facilitating second-order polymorphism. As a basis we use the simple type system based on linear logic that Selinger and Valiron have introduced into their calculus and equip it with the necessary extensions. On the one hand, none of the below mentioned λ-like calculi for quantum computation supports parametric second-order polymorphism up to now, with the exception of variants of *Lineal* ([AD08], [ADC11], [ADV11]). On the other hand, this language feature is widely used and accepted in classical programming languages and typed λ-calculi (classical in the sense of "classical computation" in contrast to "quantum computation" and not in the sense of "well-established") and there is no reason why we should not employ it in the context of quantum computation as well. However, there are indeed physical reasons (induced by the so-called *no-cloning theorem*) that force us to be more careful when modeling quantum computations than we need to be in classical settings. As a consequence we need to take appropriate measures on two different levels. Firstly, on the level of function terms we need to prohibit unrestricted copying of quantum data, since this in general contradicts physical law, and thus we would otherwise model "quantum" computations which are not physically realizable. The crucial part of this measure is the ability to expel any implicit copying such as term applications of the form $((\lambda x.f\ x\ x)\ q)$ entail during evaluation. This is the principal motivation for the wide employment of linear typing in quantum λ-calculi (or the use of similar techniques mimicking mechanisms originating in Jean-Yves Girard's *Linear Logic*, see [Gir87]). Secondly, on the level of type terms one usually marks certain types as being linear, meaning that term variables of that type are not allowed to occur more than once.[3] Mostly, only quantum data is subject to restrictions on its duplicability, and one defines a special type *Qbit* for terms representing quantum data. In order to keep function terms and the computations they embody in accordance with quantum physical law, we need to make sure this data is under all circumstances prevented from being duplicated in an unrestricted way. In the presence of parametric second-order polymorphism this means to guarantee that type term *Qbit* and type terms containing *Qbit* as a subexpression in critical positions are always marked as linear types. This is essentially what is meant by the phrase "enforceable linearity" in the title of the present work. And this concept constitutes one

[1] The book chapter [Wal05] by David Walker provides a very good introduction to the subject of so-called *substructural type systems* which are well motivated by classical programming and computing issues. The bibliographic references therein provide a valuable resource of literature for readers interested in linear typing without any reference to quantum computation.

[2] To avoid confusion, we notationally distinguish Selinger's and Valiron's *Quantum Lambda Calculus* from other quantum λ-calculi that we only mention in this introduction.

[3] Please note that we do not insist on term variables of linear type occurring *exactly* once, but adopt the less strict requirement of *at most* one occurrence. In the literature on type theory this less strict notion is sometimes referred to as affinity and thus the investigated types are sometimes called *affine types* rather than *linear types*. See [Wal05] for a classification of substructural approaches to typing.

key contribution of our work to the field of quantum programming languages. To realize this concept, we make use of already known techniques such as *subtyping* and *universal bounded quantification*. In fact, one could reuse the mechanisms that we apply to establish *enforceable linearity* for types in the *Quantum Lambda Calculus*, and doing so, one could also implement this concept (perhaps in a modified manner) in very different settings that show similar requirements. Thus, changing our point of view, we may reinterpret the present work as a case study in which we implement second-order polymorphism in a linear typing environment where *enforceable linearity* is demanded. And for this case study we have chosen Selinger's and Valiron's *Quantum Lambda Calculus*.

The history of quantum programming languages is a comparatively short one. However, a whole bunch of quantum programming languages and related calculi have been proposed within only a couple of years, starting from a first attempt by Emanuel Knill in [Kni96] to introduce conventions for quantum pseudocode, and gaining further speed in the beginning of the new millennium. Very recommendable surveys have already been given by Peter Selinger in [Sel04a] and Simon J. Gay in [Gay06]. Additional, slightly more recent resources of overview information and bibliographic references can be found in the Ph.D. theses of Benoît Valiron ([Val08]) and Margherita Zorzi ([Zor09]), for instance. In the years 2003 to 2006 there has been the annual *International Workshop on Quantum Programming Languages (QPL)* dedicated to these topics, which constitutes a pool of related works. (From 2008 on, this workshop was renamed to *International Workshop on Quantum Physics and Logic* and its scope was extended accordingly.)

The approach we base our work upon has been grounded on Selinger's influential article [Sel04b], and the first definition of Selinger's and Valiron's *Quantum Lambda Calculus* has been published in Valiron's master's thesis [Val04a]. In continuation of this line of work, there has been a series of articles: [Val04b], [SV05], [SV06], [SV08a], [SV08b], [SV09], [Val11] in which the calculus has been successively extended. The focus of these articles and Valiron's Ph.D. thesis ([Val08]) primarily lies on semantic aspects of their calculus. In the research literature we can identify at least three more lines of work that concentrate on "quantum" variants of the well-known classical λ-calculus. Two of those approaches follow similar ideas as Selinger and Valiron do, namely André van Tonder's quantum λ-calculus ([vTo04]) and the quantum λ-calculus introduced by Ugo Dal Lago, Andrea Masini and Margherita Zorzi ([DMZ09], [Zor09], [DMZ11]). A conceptually quite different approach towards a λ-like calculus modeling quantum computations has been started by Pablo Arrighi and Gilles Dowek ([AD04], [AD05], [AD08]). An outstanding peculiarity of their *Linear-algebraic Lambda Calculus* (short: *Lineal*) is that its syntax allows superpositions $\alpha.t + \beta.u$ of terms, where α and β denote complex scalars. The declared goal of their ongoing line of research (in cooperation with other authors) is to establish an appropriate type system for *Lineal* to eventually obtain (via a Curry-Howard correspondence) a logic capturing the essence of quantum computation ([ADC11], [ADV11], [DAG11]).

The present work is structured as follows. To get acquainted with the formalisms underlying quantum computation, we start off in section 2 with an introduction to the field. For the sake of brevity we confine the presentation to the required minimum that is necessary for an understanding of the subsequent sections. The no-cloning theorem is presented in this foundational part, and it plays a significant role throughout subsequent sections since it motivates the use of linear typing and the need for enforceable linearity. In section 3 we present an untyped fragment of Selinger's and Valiron's *Quantum Lambda Calculus*. In particular, we introduce a syntactically modified variation equipped with accordingly adapted operational semantics. However, all performed alternations are of a superficial nature and leave the underlying principles and mechanisms completely intact. The main part of our work is concentrated in section 4, and we therein develop our polymorphic type system for the *Quantum Lambda Calculus*. This section quite neatly divides into two parts: at first we define type terms, introduce derivation rules for well-formed types and study key consequences of our definitions. In particular, we widely treat types in isolation, and thus the obtained results related to types and type derivations are largely independent of the *Quantum Lambda Calculus*. In fact, this is the part where our major technical results are developed. Afterwards we come to the typed calculus and establish polymorphic typing in the realm of function terms. In this way we arrive at the point where we can derive well-formed function terms equipped with appropriate types. In the last part of our investigations of the polymorphically typed calculus we take a look at type safety – a key feature of typed functional programming languages and calculi.

For different reasons we have moved some contents to appendices A to D. While appendices A and B

contain supplementary technical material that has been moved there to keep the main text concise, we find complicated derivations of typed function terms in appendix C since we need horizontal format for presentation. For quick reference the last appendix section D is devoted to a complete collection of all relevant derivation rules that we encounter throughout our discourse.

The main contributions of the present work are the enrichment of simply typed QLC with explicit parametric second-order polymorphism and a detailed study of the resulting characteristics. Among the developed results are a proof of enforceable linearity with respect to type constant $Qbit$, a weak transitivity result concerning derivations of subtyping statements and concluding investigations regarding type safety and linearity.

2 A minimal quantum computation primer

Before we jump into the details of a concrete calculus modeling computations that exploit quantum phenomenons, we take a brief look at the basic formalisms that are usually used in the field of quantum computation. Since the focus of the present work lies on issues connected with type systems for such a calculus, we restrict the presentation of the foundations of quantum computation to the parts that are necessary to understand the *Quantum Lambda Calculus* (*QLC* for short), its operational semantics and the form of its type system. In particular, we present a central theorem which motivates most of the peculiarities of *QLC*'s type system in its simple version and in its polymorphic variant.

To circumvent all the cumbersome details of the standard framework for quantum computation (namely linear algebra on finite-dimensional Hilbert spaces, quantum physics' postulates, observables for measurement of quantum physical states, quantum gates, etc.), we take a somewhat spartan (but still sufficient) algebraic point of view, leaving out a lot of details and quite a few of the subtleties inherent to the standard framework. The interested reader will find the full details in introductory literature on quantum computation, for instance in [Gru99], [NC00] and [NO08].

To prepare ourselves for the formal considerations in the following subsections, we shall get acquainted with a notion that is of central importance in the rest of this work (and in quantum computation in general), namely the notion of a *qubit*. Computer scientists are very familiar with classical bits as elementary carriers of information.[4] For considerations in the theory of computation we usually regard a bit as an abstract object that can take on values 0 or 1. A qubit, on the other hand, is also treated as an abstract object which can take on a *state* that may, in a certain sense, correspond to the classical values 0 or 1, and that would then be denoted by $|0\rangle$ or $|1\rangle$, respectively. But it may also take on (uncountably many) states that are combinations of $|0\rangle$ and $|1\rangle$. The most intriguing characteristic distinguishing qubit states from classical bits' values, however, is the limitation of observation. While we can (at least in principle) always gain full knowledge of the current value of a classical bit, it is impossible to find out the *exact state* a qubit is currently in, if we do not have any *a-priori* knowledge about this state. Even worse, the process of observation in general inevitably modifies the state of a qubit, and it is random in nature with respect to a probability distribution determined by the particular sort of the observation and the qubit state immediately before the observation. We will look at the mathematical details of qubit states and their observation in the next subsection.

In analogy to classical computation theory, where bits are put into sequences to form registers that store bit strings, we compose quantum registers from single qubits. These quantum registers exhibit states that correspond to combinations of bit strings.[5] Changing our point of view, we can also regard single bits and qubits as (quantum) registers of length 1.

As the classical theory of computation does not care much about the physical realization of bits and registers in real computers, we do not bother with physical realizations of qubits and quantum registers in the present work. Nielsen and Chuang comment on this approach as follows:

> "The beauty of treating qubits as abstract entities is that it gives us the freedom to construct a general theory of quantum computation and quantum information which does not depend upon a specific system for its realization." ([NC00], page 13)

2.1 Basics from linear algebra

In the following we conceive states of quantum registers of length $n \geq 1$ as unit vectors of a 2^n dimensional Hilbert space $\mathcal{H}_n$ over the field $\mathbb{C}$ of complex numbers. Traditionally (following Paul Dirac), the state of a quantum system is denoted as a ket vector, e.g. as $|\varphi\rangle$.[6]

[4] Here, we do *not* refer to the meaning of *bit* as measure of information content in the sense of Shannon's information theory or as measure of the amount of data a device can store, for instance.

[5] This is doubtless a strongly simplified view. A very recommendable more accurate guide to these fundamental concepts is the first chapter in [NC00].

[6] In the present work this is nothing more than a notational convention without any deeper meaning. In the standard framework of quantum computation (and quantum physics in general), however, ket vectors $|\varphi\rangle$ are complemented with bra vectors $\langle\varphi|$, which together yield the quite comfortable *braket* notation, which is also reflected in the notation we use for scalar products. The interested reader will inevitably encounter these notations in the literature on quantum computation (from introductory to research level).

Definition 2.1 (Hilbert space).
A *Hilbert space* $\mathcal{H}$ is a complete vector space equipped with a scalar product, denoted $\langle\cdot|\cdot\rangle$, and a norm $\|\cdot\|$, defined as $\|\,|\varphi\rangle\,\| := \sqrt{\langle\varphi|\varphi\rangle}$ for all $|\varphi\rangle \in \mathcal{H}$. (See, for example, the appendix of [Gru99] for full mathematical details.)

Thus, a state of a quantum register (of finite length) corresponds to a vector $|\varphi\rangle \in \mathcal{H}$ in an associated (finite dimensional) Hilbert space $\mathcal{H}$ with $\|\,|\varphi\rangle\,\| = 1$.

For the rest of this work, we implicitly mean *finite dimensional Hilbert space over the field* $\mathbb{C}$ *of complex numbers* whenever we talk about a Hilbert space. Moreover, for convenience, we use the same symbol $\mathcal{H}$ for the Hilbert space $\mathcal{H}$ and also the carrier set of $\mathcal{H}$.

Definition 2.2 (scalar product).
A *scalar product* on a vector space $\mathcal{H}$ (over the field $\mathbb{C}$) is a mapping $\langle\cdot|\cdot\rangle : \mathcal{H} \times \mathcal{H} \to \mathbb{C}$ which satisfies

- $\langle\varphi|\psi\rangle = \langle\psi|\varphi\rangle^*$,
- $\langle\varphi|\varphi\rangle \geq 0$ and
  $\langle\varphi|\varphi\rangle = 0$ if and only if $|\varphi\rangle = \mathbb{0}$ is the unique zero vector in $\mathcal{H}$, and
- $\big\langle\varphi \,\big|\, \alpha\,|\psi\rangle + \beta\,|\chi\rangle\big\rangle = \alpha\langle\varphi|\psi\rangle + \beta\langle\varphi|\chi\rangle$

for arbitrary $|\varphi\rangle, |\psi\rangle, |\chi\rangle \in \mathcal{H}$ and $\alpha, \beta \in \mathbb{C}$. Here, $\cdot^*$ denotes complex conjugation, i.e. $(a+ib)^* := a - ib$.

Please note that we most of the time wrote $\langle\varphi|\psi\rangle$ instead of $\big\langle|\varphi\rangle \,\big|\, |\psi\rangle\big\rangle$ which would formally be more accurate but is not very comfortable. We stick to the more comfortable variant in our considerations.

Definition 2.3 (orthogonal and orthonormal states).
We call two states $|\varphi\rangle, |\psi\rangle \in \mathcal{H}$

- *orthogonal*, if $\langle\varphi|\psi\rangle = 0$, and
- *orthonormal*, if $|\varphi\rangle$ and $|\psi\rangle$ are orthogonal and it additionally holds $\|\,|\varphi\rangle\,\| = \|\,|\psi\rangle\,\| = 1$.

A set of vectors is called *orthogonal* (*orthonormal*), if all of its elements are pairwise orthogonal (orthonormal).

We often describe the state of quantum registers of length n as superpositions, i.e. linear combinations (with complex coefficients $\alpha_{b_j} \in \mathbb{C}$)

$$\sum_{j=0}^{2^n-1} \alpha_{b_j}\,|b_j\rangle \qquad\qquad \text{with} \quad \sum_{j=0}^{2^n-1} |\alpha_{b_j}|^2 = 1 \ ,$$

of the vectors of the *standard basis* $\mathfrak{B}_n := \{|b_0\rangle, \ldots, |b_{2^n-1}\rangle\}$. The standard basis is defined to be an orthonormal basis of $\mathcal{H}_n$, where the "names" $b_0, \ldots, b_{2^n-1} \in \{0,1\}^{(n)}$ of the basis vectors are bit strings encoding their indices in binary notation.[7] Here is an example to illustrate this for the case of 2^2 dimensions:

$$\mathfrak{B}_2 = \{|b_0\rangle := |00\rangle, |b_1\rangle := |01\rangle, |b_2\rangle := |10\rangle, |b_3\rangle := |11\rangle\} \ .$$

It turns out that the notation of basis vectors as bit strings is quite comfortable. We interpret a vector $|x\rangle = |x_0 \ldots x_{n-1}\rangle \in \mathcal{H}_n$ with $x_0, \ldots, x_{n-1} \in \{0,1\}$ as an abbreviation for the tensor product $|x_0\rangle \otimes \cdots \otimes |x_{n-1}\rangle$, where $|x_0\rangle, \ldots, |x_{n-1}\rangle \in \mathcal{H}_1$. In the same way $|x, y\rangle$ abbreviates $|x\rangle \otimes |y\rangle$, where x and y may denote bit strings (possibly of different length).[8]

Here, we leave out how the tensor product actually works on vectors in Hilbert spaces. For our purposes it is enough to know that we use it to construct states of quantum registers of length ≥ 2. We can define the scalar product on states $|\varphi_1\rangle \otimes |\psi_1\rangle = |\varphi_1, \psi_1\rangle$ and $|\varphi_2\rangle \otimes |\psi_2\rangle = |\varphi_2, \psi_2\rangle$ by

$$\langle\varphi_1, \psi_1|\varphi_2, \psi_2\rangle := \langle\varphi_1|\varphi_2\rangle\langle\psi_1|\psi_2\rangle \ . \tag{1}$$

[7] At this point we can (informally) think of $|\cdot\rangle$ as a family of bijective mappings of bit strings of a certain length n into a respective standard basis of dimension 2^n, i.e. $|\cdot\rangle = (|\cdot\rangle_n)_{n\in\mathbb{N}\setminus\{0\}}$, where the functions $|\cdot\rangle_n : \{0,1\}^{(n)} \to \mathfrak{B}_n$ do respect the tensor product of vectors, i.e. $|x_1 \ldots x_n\rangle_n = |x_1\rangle_1 \otimes \cdots \otimes |x_n\rangle_1$. We use this to get a handy notation for basis vectors.

[8] In general, for $m, n \geq 1$, the dimension of a tensor product behaves as follows:
$|\varphi\rangle \in \mathcal{H}_n \wedge |\psi\rangle \in \mathcal{H}_m \Longrightarrow |\varphi\rangle \otimes |\psi\rangle \in \mathcal{H}_{n+m}$.

In the following, we will also see that the tensor product can be used to construct mappings on Hilbert spaces of dimension ≥ 4.

Definition 2.4 (linear and unitary operators).
We define *linear mappings* (or *linear operators*) $M : \mathcal{H} \to \mathcal{H}'$ as mappings between Hilbert spaces $\mathcal{H}$ and $\mathcal{H}'$ that are compatible with vector addition and scalar multiplication, i.e. $M(\alpha\,|\varphi\rangle + \beta\,|\psi\rangle) = \alpha(M|\varphi\rangle) + \beta(M|\psi\rangle)$ for arbitrary $|\varphi\rangle, |\psi\rangle \in \mathcal{H}$. In other words, linear mappings are homomorphisms between Hilbert spaces.

A special subset of these linear mappings, namely the length preserving automorphisms, are called *unitary operators*. Thus, the inverse mapping $U^{-1} : \mathcal{H} \to \mathcal{H}$ of a unitary operator $U : \mathcal{H} \to \mathcal{H}$ always exists, and in particular, unitary operators U preserve length of vectors, i.e. $\| \, |\varphi\rangle \, \| = \|U\,|\varphi\rangle\,\|$ for all $|\varphi\rangle \in \mathcal{H}$.

The property of length preservation is a crucial one which makes unitary operators the proper tool for the transformation of the state of a quantum register into another state, since both are represented by unit vectors in the same Hilbert space. A consequence of length preservation, however, is the invertibility of unitary operators.

Unitary operators on a Hilbert space $\mathcal{H}_n$ form a group with respect to composition, i.e. the composition of two unitary operators $U, U' : \mathcal{H}_n \to \mathcal{H}_n$ is the unitary operator $UU' : \mathcal{H}_n \to \mathcal{H}_n\,;\, |\varphi\rangle \mapsto U(U'|\varphi\rangle)$. Furthermore, the tensor product of two unitary operators $U : \mathcal{H}_n \to \mathcal{H}_n$ and $V : \mathcal{H}_m \to \mathcal{H}_m$, with $n, m \geq 1$, is again a unitary operator $U \otimes V : \mathcal{H}_{n+m} \to \mathcal{H}_{n+m}$ on a Hilbert space of dimension 2^{n+m}. When we apply $U \otimes V$ to a quantum state $|\varphi\rangle \otimes |\psi\rangle \in \mathcal{H}_{n+m}$ (where $|\varphi\rangle \in \mathcal{H}_n$ and $|\psi\rangle \in \mathcal{H}_m$), we obtain

$$(U \otimes V)(|\varphi\rangle \otimes |\psi\rangle) = U\,|\varphi\rangle \otimes V\,|\psi\rangle \ .$$

Later we will use operator composition and tensor products of unitary operators to construct more complex unitary operators. For (iterated) tensor products of identical operators we may use a comfortable exponential notation, where for instance $U^{\otimes n}$ means $U \otimes \cdots \otimes U$ with n occurrences of U.

It is a fundamental postulate of quantum mechanics that the evolution of a *closed quantum system*[9] between two points in time t_1 and t_2 can be described by a unitary operator U. That means, the state $|\varphi\rangle$ at time t_1 evolves to $U\,|\varphi\rangle$ at time t_2. In particular, this gives us the tool to describe the evolution of single qubits as well as complete quantum registers.

Here, we just accept this as a fact without going into further detail. The interested reader will find more detailed information in introductory texts such as [NC00].

2.2 The no-cloning theorem

We now present a central result which constitutes one key difference of quantum information processing in contrast to its classical counterpart. In the classical setting we are usually free to copy (or *clone*, to use another word) all the information a certain information carrier embodies and transfer it to another information carrier so that we eventually obtain two distinct entities carrying precisely the same information. In the world of quantum computation, however, this dramatically changes: we cannot find a realizable physical process which transfers the complete information content of a quantum information carrier, the *source*, to another one without possibly destroying the information in the source (as long as we do not have any a-priori knowledge on the initial information in the source).

Theorem 2.5 (no-cloning theorem, due to Wootters and Zurek ([WZ82]); and Dieks ([Die82])).
Let $\mathcal{H}$ be a Hilbert space and $|c\rangle \in \mathcal{H}$ be an arbitrary fixed quantum state (used as an auxiliary state). There is no unitary operator $U : \mathcal{H} \otimes \mathcal{H} \to \mathcal{H} \otimes \mathcal{H}$ such that

$$U\,|\varphi, c\rangle = |\varphi, \varphi\rangle$$

holds for arbitrary quantum states $|\varphi\rangle \in \mathcal{H}$. In other words: a completely unknown quantum state cannot be cloned (using unitary operators).

[9] In this context "closed" means that the system does not interact with the rest of the world.

Proof. (adapted from Gruska's textbook [Gru99], proof 2 of Theorem 2.1.28, page 68)
We prove this by contradiction. Let $|\varphi\rangle$ and $|\psi\rangle$ be arbitrary quantum states (which we intend to clone) in a Hilbert space $\mathcal{H}$. Suppose we have a unitary operator $U_c : \mathcal{H} \otimes \mathcal{H} \to \mathcal{H} \otimes \mathcal{H}$ such that $U_c\,|\varphi, c\rangle = |\varphi, \varphi\rangle$ and $U_c\,|\psi, c\rangle = |\psi, \psi\rangle$ for some fixed quantum state $|c\rangle$. (Please recall, we said in the beginning we conceive quantum states as unit vectors in a Hilbert space $\mathcal{H}$, i.e. $\| \, |c\rangle \, \| = \sqrt{\langle c|c\rangle} = 1$.) Thus, we have

$$\begin{aligned}\langle\varphi|\psi\rangle = \langle\varphi|\psi\rangle \underbrace{\langle c|c\rangle}_{=1} &\overset{(1)}{=} \langle\varphi, c|\psi, c\rangle = \big\langle U_c\,|\varphi, c\rangle \;\big|\; U_c\,|\psi, c\rangle\big\rangle \\ &= \langle\varphi, \varphi|\psi, \psi\rangle \overset{(1)}{=} \langle\varphi|\psi\rangle\langle\varphi|\psi\rangle \;,\end{aligned}$$

which means $\langle\varphi|\psi\rangle = \langle\varphi|\psi\rangle\langle\varphi|\psi\rangle$ for short.[10] Hence, we may conclude either $\langle\varphi|\psi\rangle = 0$ or $\langle\varphi|\psi\rangle = 1$ holds.

It thus follows that if two quantum states $|\varphi\rangle$ and $|\psi\rangle$ are cloneable by a single unitary operator U_c, they are either orthogonal (case $\langle\varphi|\psi\rangle = 0$) or identical (case $\langle\varphi|\psi\rangle = 1$).[11] Obviously, this leads to a contradiction since we assumed U_c were capable of cloning arbitrary quantum states, in particular ones that are not necessarily orthogonal (or identical) to some distinguished quantum state. □

The no-cloning theorem entails there is no general copying mechanism (based on unitary operators) that is able to copy all the information encoded in an unknown state of one quantum bit to another quantum bit (keeping the first one intact).

However, given a specific quantum state $|\varphi\rangle \in \mathcal{H}_n$, we may construct a unitary operator which is capable of cloning $|\varphi\rangle$ and each quantum state $\big|\varphi^\perp\big\rangle \in \mathcal{H}_n$ which is orthogonal to $|\varphi\rangle$, i.e. with $\langle\varphi|\varphi^\perp\rangle = 0$. For such a construction, we first choose a basis vector $|b\rangle \in \mathfrak{B}_n$ and then compose the following four unitary operators

- $R_{\varphi\mapsto b} : \mathcal{H}_n \to \mathcal{H}_n$, mapping $|\varphi\rangle$ to basis vector $|b\rangle \in \mathfrak{B}_n$ of $\mathcal{H}_n$, and each of the orthogonal vectors in $\big\{\big|\varphi^\perp\big\rangle \;\big|\; \big\langle\varphi^\perp \;\big|\; \varphi\big\rangle = 0\big\}$ to the remaining basis vectors in $\mathfrak{B}_n \setminus \{|b\rangle\}$, such that all $R_{\varphi\mapsto b}\,\big|\varphi^\perp\big\rangle$ are pairwise distinct and distinct from $R_{\varphi\mapsto b}\,|\varphi\rangle$;[12]
- its inverse operator $R_{b\mapsto\varphi} := R^{-1}_{\varphi\mapsto b}$;
- $U^b_{|0\rangle} : \mathcal{H}_n \otimes \mathcal{H}_n \to \mathcal{H}_n \otimes \mathcal{H}_n$, mapping $|b'\rangle \otimes |0\rangle$ to $|b'\rangle \otimes |b'\rangle$ for all $|b'\rangle \in \mathfrak{B}_n$ (it is not so important how $U^b_{|0\rangle}$ works on the remaining basis vectors in $(\mathfrak{B}_n \otimes \mathfrak{B}_n) \setminus (\mathfrak{B}_n \otimes \{|0\rangle\})$, as long as $U^b_{|0\rangle}$ is unitary);[13] and
- $Id_n : \mathcal{H}_n \to \mathcal{H}_n$ is the identity mapping on $\mathcal{H}_n$; so that

$$\begin{aligned}(R_{b\mapsto\varphi} \otimes R_{b\mapsto\varphi})\; U^b_{|0\rangle}\; (R_{\varphi\mapsto b} \otimes Id_n)\,|\varphi, 0\rangle &= (R_{b\mapsto\varphi} \otimes R_{b\mapsto\varphi})\; U^b_{|0\rangle}\;\; |b, 0\rangle \\ &= (R_{b\mapsto\varphi} \otimes R_{b\mapsto\varphi})\;\; |b, b\rangle \\ &= |\varphi, \varphi\rangle \;\;.\end{aligned}$$

[10] That unitary operators U preserve the scalar product of two unit vectors $|\varphi'\rangle$ and $|\psi'\rangle$, i.e. $\langle\varphi'|\psi'\rangle = \langle U|\varphi'\rangle \,|\, U|\psi'\rangle\rangle$ with $\| \, |\varphi'\rangle \, \| = \| \, |\psi'\rangle \, \| = 1$, is a consequence of the axioms of scalar products, the definition of the norm $\| \cdot \|$, and the fact that unitary operators are defined to be length preserving. From examining the equality $\| \, |\varphi'\rangle + |\psi'\rangle \, \| = \|U(|\varphi'\rangle + |\psi'\rangle)\|$, we get that the real part of $\langle\varphi'|\psi'\rangle$ is equal to that of $\langle U\,|\varphi'\rangle \,|U\,|\psi'\rangle\rangle$, and from an examination of the equality $\| \, |\varphi'\rangle - |\psi'\rangle \, \| = \|U(|\varphi'\rangle - |\psi'\rangle)\|$, we conclude that the imaginary part of $\langle\varphi'|\psi'\rangle$ equals that of $\langle U|\varphi'\rangle \,|\, U|\psi'\rangle\rangle$.

[11] The conclusion that $|\varphi\rangle$ and $|\psi\rangle$ are identical in case of $\langle\varphi|\psi\rangle = 1$ is valid due to the Cauchy-Schwarz inequality $|\langle\varphi|\psi\rangle| \leq \| \, |\varphi\rangle \, \| \, \| \, |\psi\rangle \, \|$ (which itself is a consequence of the axioms of vector spaces and scalar products). This inequality turns into an equality if and only if we have the special case of linearly dependent vectors $|\varphi\rangle$ and $|\psi\rangle$. And since we consider $|\varphi\rangle$ and $|\psi\rangle$ to be quantum states, they have length $\| \, |\varphi\rangle \, \| = \| \, |\psi\rangle \, \| = 1$. Furthermore, we know that $|\varphi\rangle$ and $|\psi\rangle$ point in the same direction, since otherwise we would have $\langle\varphi|\psi\rangle < 0$ because of $\big\langle\varphi \;\big|\; -|\varphi\rangle\big\rangle = -\langle\varphi|\varphi\rangle \leq 0$ (due to the axioms of scalar products).

[12] Since $R_{\varphi\mapsto b}$ is defined to be a linear mapping, it is already fully specified as soon as we define its effect on a basis of $\mathcal{H}_n$. This is due to the fact that every vector $|\psi\rangle \in \mathcal{H}_n$ can be written as linear combination of basis vectors, and that linear mappings are compatible with vector addition and scalar multiplication.

[13] Please note that the construction of $U^b_{|0\rangle}$ does not contradict Theorem 2.5, since for all $\big|b_1, b'_1\big\rangle, |b_2, b'_2\rangle \in \mathfrak{B}_n \otimes \mathfrak{B}_n$ it holds $\langle b_1, b'_1|b_2, b'_2\rangle = \langle b_1|b_2\rangle\langle b'_1|b'_2\rangle$. This is because $\langle b_1|b_2\rangle$ and $\langle b'_1|b'_2\rangle$ are either 0 or 1, since all vectors in $\mathfrak{B}_n$ are orthonormal.

2.3 The role of measurement

One of the most remarkable (and puzzling) properties of quantum registers is that we in general cannot observe what state a register is exactly in at a certain point in time.

To get an idea of what the state of a quantum register might be, we need to perform a measurement on that register. With one measurement of a register in a state $|\varphi\rangle \in \mathcal{H}$ ($\mathcal{H}$ being an appropriate Hilbert space), we can distinguish an orthogonal set of states from $\mathcal{H}$.[14] Regarding the outcome of the measurement, the process of measuring is inherently random, and perhaps even more striking, the state of the measured system may change as a side effect of the measurement, and we do in general not have any chance to prevent such changes. In the end, the measured result and the state the measured system takes on immediately after the measurement are both determined randomly, depending on the former state and the exact nature of the performed measurement.

For our purposes we restrict ourselves to measurements with respect to the standard basis, as illustrated in the following subsections and in appendix A.

The simplest example

The state of a quantum register of length 1 (containing a single qubit) can be fully described by $|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ for proper $\alpha, \beta \in \mathbb{C}$ with $|\alpha|^2 + |\beta|^2 = 1$. In this case the standard basis is $\mathfrak{B}_1 = \{|0\rangle, |1\rangle\}$. Measurement of $|\varphi\rangle$ with respect to $\mathfrak{B}_1$ then yields

- either 0 with probability $|\alpha|^2$, leaving the register in state $|\varphi'\rangle = |0\rangle$,
- or 1 with probability $|\beta|^2$, leaving the register in state $|\varphi'\rangle = |1\rangle$.

Quantum registers of arbitrary length

As before, we can fully describe the state $|\varphi\rangle$ of a quantum register of length n as a superposition of basis vectors from $\mathfrak{B}_n$ of the form

$$|\varphi\rangle = \sum_{j=0}^{2^n-1} \alpha_{b_j}\,|b_j\rangle \qquad \text{with} \quad \sum_{j=0}^{2^n-1} |\alpha_{b_j}|^2 = 1\ ,$$

where $\alpha_{b_j} \in \mathbb{C}$ for all $j \in \{0, \ldots, 2^n - 1\}$. Again, measurement of $|\varphi\rangle$ with respect to $\mathfrak{B}_n$ (which corresponds to an all-at-once measurement of the complete register) yields one of the bit strings $b_j \in \{b_0, \ldots, b_{2^n-1}\} = \{0,1\}^{(n)}$ as result with probability $|\alpha_{b_j}|^2$, respectively, and leaves the register in state $|\varphi'\rangle = |b_j\rangle$, also respectively.

Perhaps more interesting, we can also measure a single qubit of the quantum register with respect to $\mathfrak{B}_1$. Measuring the qubit in the k-th position ($1 \le k \le n$) of a register in state $|\varphi\rangle = \sum_{|b\rangle \in \mathfrak{B}_n} \alpha_b\,|b\rangle$ yields

- either 0 with probability

$$p_k(0) = \sum_{b \in \mathcal{B}_k^n(0)} |\alpha_b|^2\ ,$$

  leaving the register in state

$$|\varphi'\rangle = \frac{1}{\sqrt{p_k(0)}} \sum_{b \in \mathcal{B}_k^n(0)} \alpha_b\,|b\rangle\ ,$$

- or 1 with probability

$$p_k(1) = \sum_{b \in \mathcal{B}_k^n(1)} |\alpha_b|^2\ ,$$

[14] This is a simplified view. In fact, we can more generally distinguish outcomes associated to orthogonal subspaces of $\mathcal{H}$ (i.e. Hilbert spaces $\mathcal{G} \subset \mathcal{H}$ and $\mathcal{G}' \subset \mathcal{H}$ with $\langle\varphi|\varphi'\rangle = 0$ for all $|\varphi\rangle \in \mathcal{G}$ and $|\varphi'\rangle \in \mathcal{G}'$). We use such orthogonal subspaces, for instance, when we measure a single qubit in a quantum register of length > 1.

leaving the register in state

$$|\varphi'\rangle = \frac{1}{\sqrt{p_k(1)}} \sum_{b \in \mathcal{B}_k^n(1)} \alpha_b \, |b\rangle \ ,$$

where $\mathcal{B}_k^n(x) := \{uxv \mid u \in \{0,1\}^{(k-1)} \text{ and } v \in \{0,1\}^{(n-k)}\}$ is the set of words of length n over alphabet $\{0,1\}$, where the k-th position is $x \in \{0,1\}$.

When we introduce the *Quantum Lambda Calculus* in the next section, we will see that in this framework only measurements of single qubits are possible. However, we can simulate an all-at-once measurement of a quantum register of length n with respect to $\mathfrak{B}_n$ by n consecutive measurements of all single qubits in the register with respect to $\mathfrak{B}_1$. This step-by-step measurement process eventually yields the same result with exactly the same probability leaving the register in an equivalent state after the measurement as a corresponding all-at-once measurement would do. Moreover, it turns out that it does not matter in which order the single qubits are measured. To make this clear, we develop the details of this equivalence in appendix A.

3 Untyped Quantum Lambda Calculus

The *Quantum Lambda Calculus* (short: *QLC*) was developed by Benoît Valiron and Peter Selinger in a series of works ([Val04a], [Val04b], [SV05], [SV06], [SV08a], [SV08b], [Val08], [SV09]) between 2004 and 2009. They have chosen the paradigm of *classical control, quantum data*, based on Knill's *QRAM* model of quantum computation which was informally introduced in [Kni96].[15] Peter Selinger describes this paradigm in the introduction of [Sel04b] as follows

> "[...] the data that is manipulated by programs may involve quantum superpositions, but the control state of a program is always classical; there is no 'quantum branching' and no notion of executing a quantum superposition of two different statements." ([Sel04b], page 527)

This is in contrast to the quantum Turing machine as it was originally defined by David Deutsch in [Deu85], where both is possible: superpositions of tape states and superpositions of control states of the quantum Turing machine (including the head's position on the tape).

Regarding syntax and operational semantics, untyped *QLC* is an extension of the well-known classical untyped λ-calculus (introduced by Alonzo Church around the 1930s), as we will see soon.

The next sections on syntax and operational semantics of untyped *QLC* are strongly based on [SV09]. Nevertheless, we introduce a syntax from which some syntactic sugar has been removed in order to get a clearer view of what is going on underneath, and the presentation of the operational semantics is adjusted accordingly. The performed changes, however, only involve the shape of syntactic expressions, reduction rules and congruence rules. The underlying principles and core ideas remain untouched. (We will point out the exact changes and discuss their details in due course.)

For a simplification of matters, we leave out list data structures (which are a part of *QLC* since [SV09]).

3.1 Syntax

Definition 3.1 (term variables, term constants, *QLC* function terms).

We define $\mathcal{V}_{term}$ to be the countably infinite set $\mathcal{V}_{term} := \{x, y, z, f, g, x_1, q_1, x_2, q_2, \dots\}$ of *term variables*, and $\mathcal{C}_{term}$ to be the set $\mathcal{C}_{term} := \{\text{new}, \text{meas}\} \cup \mathcal{U}$ of *term constants* with $\mathcal{U} := \bigcup_{n\geq 1} \mathcal{U}_n$ and where every $\mathcal{U}_n$ collects term constants, each of which represents a unitary operator on a Hilbert space $\mathcal{H}_n$.

We inductively define the set $\mathcal{T}_{uQLC}$ of *untyped QLC function terms* (or just *function terms*) as follows

- term variables:
  $x \in \mathcal{T}_{uQLC}$ for all $x \in \mathcal{V}_{term}$,
- term constants:
  $c \in \mathcal{T}_{uQLC}$ for all $c \in \mathcal{C}_{term}$,
- term abstraction:
  $(\lambda x.t) \in \mathcal{T}_{uQLC}$ for all $x \in \mathcal{V}_{term}$ and $t \in \mathcal{T}_{uQLC}$,
- term application:
  $(s\ t) \in \mathcal{T}_{uQLC}$ for all $s, t \in \mathcal{T}_{uQLC}$,
- pair term:
  $\langle t_1, t_2 \rangle \in \mathcal{T}_{uQLC}$ for all $t_1, t_2 \in \mathcal{T}_{uQLC}$,
- the empty tuple:
  $\langle\rangle \in \mathcal{T}_{uQLC}$,
- pair abstraction:
  $\big(\lambda\langle x, y\rangle.t\big) \in \mathcal{T}_{uQLC}$ for all $x, y \in \mathcal{V}_{term}$ with $x \neq y$ and $t \in \mathcal{T}_{uQLC}$,

[15] Knill describes the *QRAM* as "[...] a random access machine in the traditional sense with the ability to perform a restricted set of operations on quantum registers. These operations consist of state preparation, some unitary operations and measurement." ([Kni96], pages 1–2)

- disjoint union:
  $inj_l(t) \in \mathcal{T}_{uQLC}$ and $inj_r(t) \in \mathcal{T}_{uQLC}$ for all $t \in \mathcal{T}_{uQLC}$,
- case distinction:
  $\big(match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)\big) \in \mathcal{T}_{uQLC}$
  for all $x, y \in \mathcal{V}_{term}$ and $s, t_l, t_r \in \mathcal{T}_{uQLC}$,
- recursion term:
  $\big(letrec\ f = (\lambda x.s)\ in\ t\big) \in \mathcal{T}_{uQLC}$
  for all $f, x \in \mathcal{V}_{term}$ and $s, t \in \mathcal{T}_{uQLC}$.

In term abstractions $(\lambda x.t)$ we call t the *scope* of the term abstraction, and say that all free occurrences of x in t are *bound*. (We will soon give a precise definition of free term variables.) These notions are defined analogously for pair abstractions $(\lambda\langle x, y\rangle.t)$. In a recursion term $\big(letrec\ f = (\lambda x.s)\ in\ t\big)$ the *scope* of the recursion consists of function terms $(\lambda x.s)$ and t, and all free occurrences of term variable f in the scope are considered to be *bound*.

We call term constants $U \in \mathcal{U}$ *built-in unitary operators* in some places (referring to their intended meaning).

Obviously, the classical untyped λ-calculus is a (syntactic) subset of untyped *QLC*. The additional syntactic constructs have the following intuitive meaning:

- $inj_l(t)$ and $inj_r(t)$ denote the left and right inclusion (or injection) of t into a disjoint union;
- $\big(match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)\big)$ represents a case distinction depending on the form of s, where we either continue with branch $(\lambda x.t_l)$ or with the alternative branch $(\lambda y.t_r)$;
- $\big(letrec\ f = (\lambda x.s)\ in\ t\big)$ defines a recursive function $f(x) = s$ used in t;
- *new* is a function for state preparation: it takes a classical bit $b \in \{0, 1\}$ (where $0 :\equiv inj_r(\langle\rangle)$ and $1 :\equiv inj_l(\langle\rangle)$, see also the conventions below) as input, prepares a qubit in state $|b\rangle$, and returns a reference to this qubit;
- *meas* is a function for measurement: it takes a qubit, measures it with respect to the standard basis $\mathfrak{B}_1$, and returns a classical bit as result.

According to the above definition, set $\mathcal{U} = \bigcup_{n\geq 1} \mathcal{U}_n$ is a collection of term constants representing unitary operators, and these term constants are distributed among subsets $\mathcal{U}_n$ with respect to their respective arity n, i.e. with respect to the Hilbert spaces $\mathcal{H}_n$ they work on. In fact, we have different possible choices of how to define these sets $\mathcal{U}_n$.

As long as we are comfortable with an uncountable number of unitary operators, we can simply define each of the $\mathcal{U}_n$ to contain *one term constant for each possible unitary operator* U on a 2^n-dimensional Hilbert space $\mathcal{H}_n$. (Please note there exist uncountably many unitary operators on $\mathcal{H}_n$ for each $n \geq 1$.)

Envisaging physical implementations of quantum computation, it is not much of a surprise that there has been quite some effort to find sets of unitary operators that are *universal for quantum computation* (in the sense as the set $\{\neg, \wedge\}$ of logical connectives is universal for classical propositional logic, i.e. sufficient to construct all possible connectives in this logic). It is intuitively clear that this venture leads again to uncountable sets of operators as long as we pursue *exact* constructions. The set of *all* unitary one-qubit operators on $\mathcal{H}_1$ together with the so-called *controlled-not operator* $CNOT : \mathcal{H}_2 \to \mathcal{H}_2$ with

$$\begin{aligned} CNOT\,|00\rangle &= |00\rangle & CNOT\,|01\rangle &= |01\rangle \\ CNOT\,|10\rangle &= |11\rangle & CNOT\,|11\rangle &= |10\rangle \ , \end{aligned}$$

for instance, yields a universal set of unitary operators with which we can *exactly* construct all possible unitary operators on Hilbert spaces $\mathcal{H}_n$ of arbitrary finite dimension 2^n. But, as we have already said, the set $\mathcal{U}_1$ in this case becomes uncountable.

Hence, to get a universal set of unitary operators of smaller cardinality, we need to abandon the idea of *exact* constructions and turn our interest to approximations instead. It turns out that we can approximate each unitary operator (on a finite dimensional Hilbert space) up to arbitrary accuracy using a countable, even finite, set of unitary operators. One such possibility is the set of four unitary operators called Hadamard, phase, *CNOT* and $\pi/8$ gates in [NC00]. Here, we do not bother how they are defined.

The interested reader will find detailed information on universal sets of unitary quantum operators and approximation of arbitrary unitary operators for example in chapter 4 of [NC00] or in section 2.3.3 of [Gru99].

According to [SV09] the set $\mathcal{U}$ should in general consist of a fixed universal set of unitary operators. In the present work, we leave the exact choice of set $\mathcal{U}$ open, because it is not of real importance for our considerations. What is important, however, is that we partition $\mathcal{U}$ into disjoint subsets $\mathcal{U}_n$ with $n \geq 1$, as we have done in Definition 3.1.

Remark: In the above syntax definition we diverge a bit from the original in [SV09] in the way we define case distinction and recursion. As announced in the beginning of the section, we have stripped off a bit of syntactic sugar. The original ("sugared") versions look as follows:

"sugared" case distinction: $\big(match\ s\ with\ (x \mapsto t_l \mid y \mapsto t_r)\big)$ "sugared" recursion: $\big(letrec\ f\ x = s\ in\ t\big)$

for term variables $f, x, y \in \mathcal{V}_{term}$ and function terms $s, t, t_l, t_r \in \mathcal{T}_{uQLC}$.

Of course, we have to adapt all other notions accordingly in order to handle the "less sugared" syntax. What we gain from this are clearer (i.e. less ambiguous) notions of free term variables and substitution and simpler definitions. In addition, the formal definition of operational semantics will be more familiar to readers acquainted with the classical λ-calculus.

To improve readability of function terms in *QLC*, we agree on the following conventions:

$$\begin{aligned}
(\lambda x_1 \dots x_n.t) &:\equiv (\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.t)\dots))) \\
(t_1\ t_2\ \dots\ t_n) &:\equiv (\dots((t_1\ t_2)\ t_3)\dots\ t_n) \\
\langle t_1, \dots, t_n\rangle &:\equiv \langle t_1, \dots \langle t_{n-2}, \langle t_{n-1}, t_n\rangle\rangle \dots \rangle \\
0 &:\equiv inj_r(\langle\rangle) \\
1 &:\equiv inj_l(\langle\rangle) \\
\big(if\ s\ then\ t_l\ else\ t_r\big) &:\equiv \big(match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)\big) \\
&\qquad\qquad \text{(where } x \text{ and } y \text{ are fresh)}
\end{aligned}$$

Furthermore, we may drop outer parentheses when it is convenient to do so.

Definition 3.2 (free term variables, α-equivalence).
We recursively define function $ftmv : \mathcal{T}_{uQLC} \to \mathcal{P}(\mathcal{V}_{term})$ which yields the set $ftmv(t)$ of all term variables that occur as *free term variables* in function term t (for $c \in \mathcal{C}_{term}$ and $x \in \mathcal{V}_{term}$):[16]

$$\begin{aligned}
ftmv\big(c\big) &:= \emptyset\ , \\
ftmv\big(x\big) &:= \{x\}\ , \\
ftmv\big((\lambda x.t)\big) &:= ftmv(t) \setminus \{x\}\ , \\
ftmv\big((s\ t)\big) &:= ftmv(s) \cup ftmv(t)\ , \\
ftmv\big(\langle t_1, t_2\rangle\big) &:= ftmv(t_1) \cup ftmv(t_2)\ , \\
ftmv\big(\langle\rangle\big) &:= \emptyset\ , \\
ftmv\big((\lambda\langle x, y\rangle.t)\big) &:= ftmv(t) \setminus \{x, y\}\ , \\
ftmv\big(inj_l(t)\big) = ftmv\big(inj_r(t)\big) &:= ftmv(t)\ , \\
ftmv\big((match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r))\big) &:= ftmv(s) \cup ftmv((\lambda x.t_l)) \cup ftmv((\lambda y.t_r))\ , \\
ftmv\big((letrec\ f = (\lambda x.s)\ in\ t)\big) &:= \big(ftmv((\lambda x.s)) \cup ftmv(t)\big) \setminus \{f\}\ .
\end{aligned}$$

We call function terms α-*equivalent* if they differ only in the names of their bound term variables, i.e. if they have the same structure.

From now on we follow the usual convention to identify function terms that are α-equivalent.

[16] We use the notation $\mathcal{P}(S)$ to denote the power set of a set S.

Definition 3.3 (substitution of term variables).
We denote *substitution of free term variables* in *QLC* function terms as $t[t'/z]$, meaning that all free occurrences of term variable z in function term t are substituted by function term t'. The recursive definition is as follows (for $c \in \mathcal{C}_{term}$ and $x \in \mathcal{V}_{term}$):

$$
\begin{aligned}
c[t'/z] &:= c \ , \\
z[t'/z] &:= t' \ , \\
x[t'/z] &:= x \ , && \text{where } z \neq x \ ; \\
(\lambda z.t)[t'/z] &:= (\lambda z.t) \ , \\
(\lambda x.t)[t'/z] &:= (\lambda x'.t[x'/x][t'/z]) \ , && \text{where } z \neq x \text{ and} \\
& && x' \notin \mathit{ftmv}(t) \cup \mathit{ftmv}(t') \cup \{z, x\} \ ; \\
(s\ t)[t'/z] &:= (s[t'/z]\ t[t'/z]) \ , \\
\langle t_1, t_2 \rangle[t'/z] &:= \langle t_1[t'/z], t_2[t'/z] \rangle \ , \\
\langle\rangle[t'/z] &:= \langle\rangle \ , \\
(\lambda\langle x, y\rangle.t)[t'/z] &:= (\lambda\langle x, y\rangle.t) \ , && \text{where } z = x \text{ or } z = y \ ; \\
(\lambda\langle x, y\rangle.t)[t'/z] &:= (\lambda\langle x', y'\rangle.t[x'/x][y'/y][t'/z]) \ , && \text{where } x \neq z \neq y \text{ and} \\
& && \text{we choose fresh } x' \neq y' \text{ with} \\
& && x', y' \notin \mathit{ftmv}(t) \cup \mathit{ftmv}(t') \\
& && \qquad \cup \{z, x, y\} \ ; \\
\mathit{inj}_l(t)[t'/z] &:= \mathit{inj}_l(t[t'/z]) \ , \\
\mathit{inj}_r(t)[t'/z] &:= \mathit{inj}_r(t[t'/z]) \ ,
\end{aligned}
$$

$$
\begin{aligned}
\big(\mathit{match}\ s\ \mathit{with}\ (\lambda x.t_l) \mid (\lambda y.t_r)\big)[t'/z] &:= \\
&\big(\mathit{match}\ s[t'/z]\ \mathit{with}\ (\lambda x.t_l)[t'/z] \mid (\lambda y.t_r)[t'/z]\big) \ ,
\end{aligned}
$$

$$
\begin{aligned}
\big(\mathit{letrec}\ z = (\lambda x.s)\ \mathit{in}\ t\big)[t'/z] &:= \big(\mathit{letrec}\ z = (\lambda x.s)\ \mathit{in}\ t\big) \ , \\
\big(\mathit{letrec}\ f = (\lambda x.s)\ \mathit{in}\ t\big)[t'/z] &:= \big(\mathit{letrec}\ f' = (\lambda x.s)[f'/f][t'/z]\ \mathit{in}\ t[f'/f][t'/z]\big) \ , \\
& \qquad \text{where } f \neq z \text{ and we choose } f' \text{ to be fresh, i.e.} \\
& \qquad f' \notin \mathit{ftmv}((\lambda x.s)) \cup \mathit{ftmv}(t) \cup \mathit{ftmv}(t') \cup \{z, f\}.
\end{aligned}
$$

The way substitution deals with term abstractions, pair abstractions and similar constructs is taken over from [Cro93]. In case $(\lambda x.t)[t'/z]$, for instance, we rename bound term variable x to a fresh term variable x' in t before the actual substitution in performed, in order to keep free occurrences of x in t' from being bound unintendedly by the former term abstraction $(\lambda x.\ \ldots)$ after substitution of z in t.

3.2 Operational semantics

To clarify and formalize the intended computational behavior of the syntactic constructs we introduced in the previous section, we present formal reduction rules according to which *QLC* function terms are evaluated.

But first, we need to provide an adequate formalism to talk about terms with associated quantum data.

Definition 3.4 (quantum closures).
A *quantum closure* is a triple $[Q, L, t]$, where

- Q is a unit vector $|\varphi\rangle \in \mathcal{H}_n$ for some $n \geq 0$;
- L is a sequence of n distinct term variables, denoted as $|q_1, \ldots, q_n\rangle$, and called *linking sequence*;
- t is a function term from $\mathcal{T}_{uQLC}$ whose free term variables *all* appear in L, i.e. $\mathit{ftmv}(t) \subseteq \{q_1, \ldots, q_n\}$.

We call the *set of all quantum closures* $\mathcal{S}_{QLC}$ and will occasionally refer to quantum closures as *states*.[17]

A term variable q_i, $1 \leq i \leq n$, that occurs freely in function term t is considered *bound* in a quantum closure $[Q, |q_1, \ldots, q_n\rangle, t]$, where it is captured by linking sequence $L = |\ldots, q_i, \ldots\rangle$. Every free occurrence of q_i in function term t is then associated to the i-th qubit in Q, counted from left to right, and we use those occurrences of q_i in t as references to address the associated quantum data.[18]

The name "quantum closure" indicates that the embedded function terms become closed in the sense that all remaining free term variables are bound by linking sequence L in the second component.

Now that we have introduced an additional way of binding term variables in function terms embedded in quantum closures, we extend the notion of α-*equivalence* to quantum closures that differ only in their bound variables. Henceforth, we thus also identify α-equivalent quantum closures.

With quantum closures we have a tool at hand that enables us to maintain and manipulate a quantum state during the process of term evaluation. However, by introducing this concept of state, we give up some of the purity and elegance of functional approaches to programming, namely the absence of side effects of function applications, because reduction rules that manipulate the embedded quantum data in a quantum closure do mostly not reflect the performed changes in the reduced terms (as we will see soon).

From some informal examples (which are supposed to reflect our intuition of the evaluation of *QLC* function terms) it becomes clear that it is particularly important to fix a reduction strategy for evaluation of function terms in *QLC*.

Example 3.5 (taken from [SV09]).
Consider function term coin $:\equiv \big(\lambda z.\text{meas}\ (H\ (\text{new}\ 0))\big)$, where

$$H : \mathcal{H}_1 \to \mathcal{H}_1; \quad |x\rangle \mapsto \tfrac{1}{\sqrt{2}}\big(|0\rangle + (-1)^x |1\rangle\big) \quad \text{with } |x\rangle \in \mathfrak{B}_1 = \{|0\rangle, |1\rangle\}$$

is a unitary operator, the so-called *Hadamard operator*. The intuitive idea behind function term *coin* is that of a function implementing the toss of a fair "quantum coin", i.e. *coin* gives result 0 or 1 with equal probability, as soon as we apply an arbitrary argument to it. Next, we define the boolean *xor* function

$$\text{xor} :\equiv \big(\lambda xy.\mathit{if}\ x\ \mathit{then}\ (\mathit{if}\ y\ \mathit{then}\ 0\ \mathit{else}\ 1)\ \mathit{else}\ (\mathit{if}\ y\ \mathit{then}\ 1\ \mathit{else}\ 0)\big)\ .$$

Let us take a look at two exemplary reductions of function term $(\lambda x.\text{xor}\ x\ x)\ (\text{coin}\ \langle\rangle)$ where we apply different reduction orders, namely *call-by-name* and *call-by-value*, which we define in the standard way (see, for instance, [Pie02], pages 56 and 57).

For our examples we use the initial quantum closures $[\mathbb{1}, |\rangle, t]$ for *closed* function terms t (i.e. $\mathit{ftmv}(t) = \emptyset$), where $\mathbb{1}$ denotes the positive unit vector in the one-dimensional Hilbert space $\mathcal{H}_0$, and for which we have $\mathbb{1} \otimes |\varphi\rangle = |\varphi\rangle \otimes \mathbb{1} = |\varphi\rangle$ for any quantum state $|\varphi\rangle$. $|\rangle$ denotes an empty sequence of term variables.

Using a call-by-name reduction strategy, we get

$$\begin{aligned}\big[\mathbb{1}, |\rangle, (\lambda x.\text{xor}\ x\ x)\ (\text{coin}\ \langle\rangle)\big] &\to \big[\mathbb{1}, |\rangle, \text{xor}\ (\text{coin}\ \langle\rangle)\ (\text{coin}\ \langle\rangle)\big] \\ &\to^* \begin{cases} \big[|00\rangle, |q_1, q_2\rangle, 0\big] & \text{with probability } \frac{1}{4}, \\ \big[|01\rangle, |q_1, q_2\rangle, 1\big] & \text{with probability } \frac{1}{4}, \\ \big[|10\rangle, |q_1, q_2\rangle, 1\big] & \text{with probability } \frac{1}{4}, \\ \big[|11\rangle, |q_1, q_2\rangle, 0\big] & \text{with probability } \frac{1}{4}. \end{cases}\end{aligned}$$

[17] Here, we distinguish between the formally defined syntactic construct called "quantum closure" and the informally used semantic object called "state" or, to be more precise, "*evaluation state of a QLC function term with associated quantum data*" which is actually described by a quantum closure.

[18] Please note the association between term variable q_i and the i-th qubit in Q is not due to q_i's index i but rather due to the i-th position of q_i in the linking sequence L. Consider, for instance, quantum closure $[Q, |x, y, z\rangle, t]$. There, every free occurrence of x in t is associated to the first qubit in Q, free y's are associated to the second and free occurrences of z to the third qubit in Q.

In contrast, using a call-by-value reduction strategy, we get

$$\begin{aligned}
&\left[\mathbb{1}, |\rangle, (\lambda x.\text{xor } x\ x)\ (\text{coin}\ \langle\rangle)\right] \\
&\qquad \to^* \begin{cases} \left[|0\rangle, |q_1\rangle, (\lambda x.\text{xor } x\ x)\ 0\right] & \text{with probability } \frac{1}{2}, \\ \left[|1\rangle, |q_1\rangle, (\lambda x.\text{xor } x\ x)\ 1\right] & \text{with probability } \frac{1}{2}, \end{cases} \\
&\qquad \to \begin{cases} \left[|0\rangle, |q_1\rangle, \text{xor } 0\ 0\right] \to^* \left[|0\rangle, |q_1\rangle, 0\right] \\ \left[|1\rangle, |q_1\rangle, \text{xor } 1\ 1\right] \to^* \left[|1\rangle, |q_1\rangle, 0\right] \end{cases}
\end{aligned}$$

This obviously disagreeing behavior is caused by the random nature of quantum mechanical measurement in *coin* (embodied by the operator *meas*) and by the fact that the operator *new* is not idempotent with respect to its impact on the associated quantum data, and thus produces disagreeing quantum states when it is reduced differently often due to differing reduction strategies.

Example 3.6.
Suppose we intend to evaluate tuple $\langle \text{new } 0,\ \text{new } 1,\ \text{new } 0,\ \text{new } 1\rangle$. When we evaluate it from left to right, we get

$$\begin{aligned}
&\left[\mathbb{1}, |\rangle, \langle \text{new } 0,\ \text{new } 1,\ \text{new } 0,\ \text{new } 1\rangle\right] \\
&\qquad \to^* \left[|0101\rangle, |q_1, q_2, q_3, q_4\rangle, \langle q_1, q_2, q_3, q_4\rangle\right] .
\end{aligned}$$

In contrast, we obtain the following result if we evaluate the tuple from right to left:

$$\begin{aligned}
&\left[\mathbb{1}, |\rangle, \langle \text{new } 0,\ \text{new } 1,\ \text{new } 0,\ \text{new } 1\rangle\right] \\
&\qquad \to^* \left[|1010\rangle, |q_1, q_2, q_3, q_4\rangle, \langle q_4, q_3, q_2, q_1\rangle\right] .
\end{aligned}$$

Obviously, in this example the order of qubits appearing in the quantum state of the resulting quantum closure depends on the applied reduction strategy. This is due to the fact that reductions of subterms *new* x_1 and *new* x_2, where $x_1, x_2 \in \{0, 1\}$ and $x_1 \neq x_2$ do not commute with respect to the modified quantum state.

Selinger and Valiron emphasize in [SV09] that the decision which strategy to choose is largely a matter of taste. They have chosen a call-by-value reduction strategy and encode this strategy in the definition of reduction rules and congruence rules (which allow us to evaluate only certain subexpressions).

Although we also establish the *call-by-value reduction strategy* by the form of reduction and congruence rules below, it is still useful to put it in words first:

> "[...] only outermost redexes are reduced *and* [...] a redex is reduced only when its right-hand side has already been reduced to a *value* – a term that is finished computing and cannot be reduced any further." ([Pie02], page 57)

As usual, the term *redex* stands for "reducible (sub)expression" and means a function term or subterm that may be reduced by applying a reduction rule (possibly with the help of appropriate congruence rules). This will become clear later (in Definitions 3.8, 3.10 and 3.11), although we will not define this notion formally.

Actually, due to the remaining syntactic sugar, it is not always the right-hand side of a redex that needs to be fully reduced first, especially concerning the function term s in a case distinction $(match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r))$. However, when we look at the result of a reduction of such a case distinction (cf. Definition 3.8), we find s needs to be fully evaluated to a disjoint union $inj_l(s')$ or $inj_r(s')$ first, before we can apply its inner part s' to one of the abstractions $(\lambda x.t_l)$ or $(\lambda y.t_r)$, where it then becomes a right-hand side of a redex. Hence, we moved the full reduction of the right-hand side of redexes $((\lambda x.t_l)\ s')$ and $((\lambda y.t_r)\ s')$ already to an earlier stage of the evaluation. (This is indeed necessary to really define a reduction strategy, i.e. a way of unambiguously saying which redex is to be reduced next, since otherwise there might be more than one redex in a function term at a certain point in time, and we would thus introduce ambiguity into our "reduction strategy".)

Let us now continue in our formal presentation of the operational semantics and clarify how the above quoted "values" are defined in untyped *QLC*.

Definition 3.7 (value terms, value states).
The set $\mathcal{T}_{value}$ of *value terms* is inductively defined by

$$\begin{aligned} c, x, \langle\rangle &\in \mathcal{T}_{value} \text{ , where } c \in \mathcal{C}_{term} \text{ and } x \in \mathcal{V}_{term}, \\ (\lambda x.t), (\lambda\langle x, y\rangle.t) &\in \mathcal{T}_{value} \text{ , where } x, y \in \mathcal{V}_{term} \text{ and } t \in \mathcal{T}_{uQLC} \text{ ,} \\ inj_l(v), inj_r(v) &\in \mathcal{T}_{value} \text{ , where } v \in \mathcal{T}_{value} \text{ ,} \\ \langle v, w\rangle &\in \mathcal{T}_{value} \text{ , where } v, w \in \mathcal{T}_{value} \text{ .} \end{aligned}$$

A quantum closure $[Q, L, v]$ is called *value state* if v is a value term; and we call the set of all value states $\mathcal{S}_{value}$.

Obviously, the set of value terms is a (proper) subset of function terms, i.e. $\mathcal{T}_{value} \subset \mathcal{T}_{uQLC}$, and we get the same relation $\mathcal{S}_{value} \subset \mathcal{S}_{QLC}$ for the respective sets of states.

Value states can be regarded as "fully evaluated states". Thus, a value term (embedded in a quantum closure) is not evaluated any further. Looking at the above definition, we see term abstractions $(\lambda x.t)$ and $(\lambda\langle x, y\rangle.t)$ (where t stands for an arbitrary *QLC* function term) are regarded as fully evaluated, although t might still contain reducible subexpressions. We can interpret this as a function that we could simplify, if we wished to. However, *QLC*'s reduction strategy does not consider the internal structure of the scope of an abstraction (be it term abstraction or pair abstraction), as long as no argument is applied to it.[19]

Remark: In contrast to value terms in [SV09], we here have a new sort of value terms due to stripping off syntactic sugar: $(\lambda\langle x, y\rangle.t)$. But this is nothing totally new, since in the original version, we can have a value term of the form $(\lambda z.(let\ \langle x, y\rangle = z\ in\ t))$ with $z \notin ftmv(t)$, which is defined to be equal to $(\lambda\langle x, y\rangle.t)$ in Convention 1.3.2 in [SV09].

Next, we formally define reduction rules (and implicitly also fix the reduction strategy) for *QLC* function terms embedded in quantum closures. Although the rewrite procedure will be a probabilistic one that works on whole quantum closures, not all of the reduction rules take the complete quantum closure into account and only a few execute with probability smaller than 1. This justifies to take a simplified point of view for some basic reduction rules by concentrating on their effect on function terms, if and only if they leave the first two components of a quantum closure alone and exhibit a reduction probability equal to 1. Following this suggestion, we take a look at *simplified* reduction rules concerned with *QLC* function terms that implement the "classical control" part.

Definition 3.8 (basic reduction rules).
We define the following set of basic reduction rules for arbitrary $s, t, t_l, t_r \in \mathcal{T}_{uQLC}$; $v, w \in \mathcal{T}_{value}$; $x, y, f \in \mathcal{V}_{term}$:

$$\begin{aligned} (\lambda x.t)\ v &\to t[v/x] \text{ ,} \\ (\lambda\langle x, y\rangle.t)\ \langle v, w\rangle &\to t[v/x, w/y] \text{ ,} \\ match\ inj_l(v)\ with\ (\lambda x.t_l) \mid (\lambda y.t_r) &\to (\lambda x.t_l)\ v \text{ ,} \\ match\ inj_r(w)\ with\ (\lambda x.t_l) \mid (\lambda y.t_r) &\to (\lambda y.t_r)\ w \text{ ,} \\ letrec\ f = (\lambda x.s)\ in\ t &\to t\big[(\lambda x.(letrec\ f = (\lambda x.s)\ in\ s))\ /\ f\big] \text{ .} \end{aligned}$$

In the second reduction rule, we used *simultaneous substitution*, denoted as $t[v/x, w/y]$, i.e. all free occurrences of term variables x and y are substituted simultaneously in function term t, and we must have $x \neq y$. Simultaneously means that expressions v introduced by substitution $[v/x]$ are not considered by substitution $[w/y]$ and vice versa. If y does not occur as free term variable in v, this is equal to

[19] In fact, this is necessary to keep the semantics sound, since if we considered a quantum closure $[Q, |q_1\rangle, (\lambda y.U\langle y, q_1\rangle)]$, and intended to reduce the subterm $U\langle y, q_1\rangle$, we would need to consider the "quantum closure" $[Q, |q_1\rangle, U\langle y, q_1\rangle]$, similarly to the congruence rules we define later. But this is not a well-defined quantum closure, because the free term variable y does not appear in linking sequence $|q_1\rangle$. Besides this formal restriction, however, it is at this point unclear how the unitary operator U shall (semantically) act on y, since y does not refer to a qubit in Q (see Definition 3.10 for the definition of reduction of unitary operators).

$t[v/x][w/y]$, and if x does not appear freely in w it equals $t[w/y][v/x]$. Let us illustrate this by an example:

$$\begin{aligned}&\big\langle(\lambda x.t\ x\ y),\ (\lambda z.s\ x\ y\ z)\big\rangle[v/x, w/y]\\ &\quad= \big\langle(\lambda x.t\ x\ y)[v/x, w/y],\ (\lambda z.s\ x\ y\ z)[v/x, w/y]\big\rangle\\ &\quad= \big\langle(\lambda x'.t[x'/x][w/y]\ x'\ w),\ (\lambda z'.s[z'/z][v/x, w/y]\ v\ w\ z')\big\rangle\ ,\end{aligned}$$

where x' and z' are fresh term variables that have not occurred before, i.e. in particular $x', z' \notin \mathit{ftmv}(t) \cup \mathit{ftmv}(s) \cup \mathit{ftmv}(v) \cup \mathit{ftmv}(w) \cup \{x, y\}$.

The above reduction rules reflect the chosen evaluation strategy in that they allow only value terms to be substituted for free variables in function terms. This means that arguments have to be evaluated to a value term before they can be applied to a term abstraction, pair abstraction or in a case distinction.

Remark: In the case of the given basic reduction rules a comparison to [SV09] shows that the syntactic shape of all of these reduction rules diverges (more or less) from their original counterpart. Of course, this is again due to stripping off syntactic sugar. We need to convince ourselves at least informally that the changed reduction rules still lead to an equivalent operational semantics. Our first two reduction rules correspond to

$$\mathit{let}\ x = v\ \mathit{in}\ t\ \to\ t[v/x] \quad \text{and} \quad \mathit{let}\ \langle x, y\rangle = \langle v, w\rangle\ \mathit{in}\ t\ \to\ t[v/x, w/y]$$

in [SV09]. But since Convention 1.3.2 in that same work contains the identities $(\mathit{let}\ x = s\ \mathit{in}\ t) \equiv ((\lambda x.t)\ s)$ and $(\lambda\langle x, y\rangle.t) \equiv (\lambda z.(\mathit{let}\ \langle x, y\rangle = z\ \mathit{in}\ t))$ with a fresh $z \notin \mathit{ftmv}(t)$, we immediately see that our first and second reduction rules are equivalent to their counterparts in [SV09].

Reduction rules three and four are somewhat more interesting, since the corresponding reduction rules in the original work perform in one (atomic) reduction step, what we have to put into two consecutive reduction steps: reduction rule

$$\mathit{match}\ \mathit{inj}_l(v)\ \mathit{with}\ (x \mapsto t_l \mid y \mapsto t_r)\ \to\ t_l[v/x]$$

from [SV09] corresponds to the two-step reduction

$$\mathit{match}\ \mathit{inj}_l(v)\ \mathit{with}\ (\lambda x.t_l) \mid (\lambda y.t_r)\ \xrightarrow{(1)}\ (\lambda x.t_l)\ v\ \xrightarrow{(2)}\ t_l[v/x]$$

according to our basic reduction rules in Definition 3.8 (and analogously for the other case distinction rule). There are two facts which ensure that this does not lead to changed computational behavior. The first one is that v is a value term and is thus already fully evaluated. This means there cannot occur any reduction steps reducing v (or one of its subexpressions) between reduction steps (1) and (2). The other fact is that due to the fixed call-by-value reduction strategy, expression $(\mathit{match}\ \mathit{inj}_l(v)\ \mathit{with}\ \ldots)$ needs to be fully evaluated (before anything else can be evaluated), once evaluation of it has begun, and term application $((\lambda x.t_l)\ v)$ is clearly not yet fully evaluated. Hence, reduction step (2) is the one and only possible reduction step that can be performed immediately after (1). This is formally ensured by Definitions 3.8, 3.10 and 3.11.

Finally, is it not hard to see that the fifth reduction rule, which corresponds to

$$(\mathit{letrec}\ f\ x = s\ \mathit{in}\ t)\ \to\ t\big[(\lambda x.(\mathit{letrec}\ f\ x = s\ \mathit{in}\ s))\ /\ f\big]$$

from [SV09], just moves the somewhat hidden binding of x in s closer to its scope. But this has already been done in the definition of the syntax of function terms. Here, it is just taken over into the corresponding reduction rule and the rest of the rule is left unchanged.

Since we now have formally fixed how the "classical control" part of *QLC* behaves computationally, it remains to treat the part modeling quantum computations. Of course, this involves the quantum data embedded in a quantum closure and also introduces probabilistic aspects due to quantum mechanical measurement.

Definition 3.9 (probabilistic reductions of quantum closures).
To address a single *probabilistic reduction step from quantum closure* $[Q, L, t]$ *to* $[Q', L', t']$ *that is executed with probability* $p \in [0,1]$ ($[0,1]$ being the real unit interval), we write

$$[Q, L, t] \to_p [Q', L', t'] \ .$$

We use the usual *reflexive, transitive closure operator* $\to_p^*$ to abbreviate a finite sequence of single-step reductions $[Q_0, L_0, t_0] \to_{p_1} \ldots \to_{p_n} [Q_n, L_n, t_n]$ for an arbitrary $n \geq 0$, and thus write $[Q_0, L_0, t_0] \to_p^* [Q_n, L_n, t_n]$, where $p := \begin{cases} 1 & \text{, if } n = 0, \\ \prod_{i=1}^n p_i & \text{, otherwise.} \end{cases}$

It is now time to give up the simplified point of view and turn the above basic reduction rules into probabilistic reduction rules on quantum closures that we need to evaluate QLC function terms. Furthermore, we still need to give reduction rules dealing with quantum data.

Definition 3.10 (reduction of quantum closures).
For each of the basic reduction rules $s \to t$ in Definition 3.8 we have a *probabilistic reduction rule on quantum closures*

$$[Q, L, s] \to_1 [Q, L, t]$$

for all $[Q, L, s] \in \mathcal{S}_{QLC}$.

We moreover define the following *probabilistic reduction rules for initialization, manipulation and measurement of quantum data*:

$$\begin{aligned}
[\ Q, |q_1, \ldots, q_n\rangle, \text{new } 0\ ] &\to_1 [\ Q \otimes |0\rangle, |q_1, \ldots, q_n, q_{n+1}\rangle, q_{n+1}\ ] \ , \\
[\ Q, |q_1, \ldots, q_n\rangle, \text{new } 1\ ] &\to_1 [\ Q \otimes |1\rangle, |q_1, \ldots, q_n, q_{n+1}\rangle, q_{n+1}\ ] \ , \\
[\ Q, |q_1, \ldots, q_n\rangle, U_1\ q_j\ ] &\to_1 [\ Q', |q_1, \ldots, q_n\rangle, q_j\ ] \ , \\
[\ Q, |q_1, \ldots, q_n\rangle, U_{\geq 2}\ \langle q_{j_1}, \ldots, q_{j_m}\rangle\ ] &\to_1 [\ Q'', |q_1, \ldots, q_n\rangle, \langle q_{j_1}, \ldots, q_{j_m}\rangle\ ] \ , \\
[\ \alpha\, Q_0 + \beta\, Q_1, |q_1, \ldots, q_n\rangle, \text{meas } q_i\ ] &\to_{|\alpha|^2} [\ Q_0, |q_1, \ldots, q_n\rangle, 0\ ] \ , \\
[\ \alpha\, Q_0 + \beta\, Q_1, |q_1, \ldots, q_n\rangle, \text{meas } q_i\ ] &\to_{|\beta|^2} [\ Q_1, |q_1, \ldots, q_n\rangle, 1\ ] \ .
\end{aligned}$$

In the rules for the *new* operator, Q is an n-qubit state such that $Q \otimes |b\rangle$, $b \in \{0,1\}$, is an $(n+1)$-qubit state. The embedded quantum state $Q \in \mathcal{H}_n$ is thus lifted to a state $Q \otimes |b\rangle \in \mathcal{H}_{n+1}$ with a doubled number of dimensions.

In the third rule we reduce a function term $(U_1\ q_j)$, $1 \leq j \leq n$, for a unary built-in unitary operator $U_1 \in \mathcal{U}_1$. The quantum state Q from an appropriate 2^n-dimensional Hilbert space $\mathcal{H}_n$ is transformed into quantum state $Q' \in \mathcal{H}_n$, such that Q' is derived from Q by applying U_1 to the j-th qubit in Q, i.e.

$$Q' := \left(Id_1^{\otimes j-1} \otimes U_1 \otimes Id_1^{\otimes n-j}\right) Q \ ,$$

where $Id_1 : \mathcal{H}_1 \to \mathcal{H}_1;\ |\varphi\rangle \mapsto |\varphi\rangle$ denotes the identity operator on a two-dimensional Hilbert space (which is obviously unitary).[20]

In the fourth rule (which reduces a term $(U_{\geq 2}\ \langle q_{j_1}, \ldots, q_{j_m}\rangle)$), $U_{\geq 2} \in \mathcal{U}_m$ is an m-ary built-in unitary operator with $m \geq 2$, $q_{j_1}, \ldots, q_{j_m}$ all appear in linking sequence $|q_1, \ldots, q_n\rangle$, and indices $j_1, \ldots, j_m$ are pairwise distinct. The quantum state Q'' is obtained from Q by applying $U_{\geq 2}$ to qubits $j_1, \ldots, j_m$. (What this last sentence precisely means is elaborated in more detail in appendix B.)

In the last two rules for measurement, Q_0 and Q_1 represent *normalized* quantum states (i.e. $\|Q_0\| = \|Q_1\| = 1$) of the form

$$Q_0 := \sum_j \alpha_j \left|\varphi_j^0\right\rangle \otimes |0\rangle \otimes \left|\psi_j^0\right\rangle \quad \text{and} \quad Q_1 := \sum_j \beta_j \left|\varphi_j^1\right\rangle \otimes |1\rangle \otimes \left|\psi_j^1\right\rangle ,$$

[20] Intuitively, this rule is a special case of the fourth rule for the application of *unary* unitary operators. However, since we do not write angle brackets around a single free term variable q_j, we provide a syntactically adjusted reduction rule for this special case.

where $\left|\varphi_j^0\right\rangle, \left|\varphi_j^1\right\rangle \in \mathcal{H}_{i-1}$ are $(i-1)$-qubit states, and $\left|\psi_j^0\right\rangle, \left|\psi_j^1\right\rangle \in \mathcal{H}_{n-i}$ are $(n-i)$-qubit states, such that the i-th qubit in Q, represented by q_i, is the one measured during these reductions.

Please note that, by definition of quantum closures, $\alpha Q_0 + \beta Q_1$ in the last two reduction rules needs to be a unit vector, that is $\|\alpha Q_0 + \beta Q_1\| = 1$ must hold. As a consequence of $\|\alpha Q_0 + \beta Q_1\| = 1$, the required $\|Q_0\| = \|Q_1\| = 1$ and the fact that Q_0 and Q_1 are orthogonal (which follows from their above definition), we get $|\alpha|^2 + |\beta|^2 = 1$, since

$$\begin{aligned} 1^2 \;&=\; \|\alpha Q_0 + \beta Q_1\|^2 \;=\; \langle \alpha Q_0 + \beta Q_1 \mid \alpha Q_0 + \beta Q_1 \rangle \\ &=\; \alpha\alpha^* \underbrace{\langle Q_0|Q_0\rangle}_{=\,\|Q_0\|^2=1} + \alpha\beta^* \underbrace{\langle Q_1|Q_0\rangle}_{=\,0} + \beta\alpha^* \underbrace{\langle Q_0|Q_1\rangle}_{=\,0} + \beta\beta^* \underbrace{\langle Q_1|Q_1\rangle}_{=\,\|Q_1\|^2=1} \;=\; |\alpha|^2 + |\beta|^2 \;. \end{aligned}$$

Hence, these two reduction rules properly model measurement of orthogonal subspaces of $\mathcal{H}_n$.

Moreover, when we take a closer look at the left-hand quantum closures in the reduction rules from Definition 3.10, we see that the function terms embedded in these quantum closures are term applications where value terms are applied to term constants. This is in full accordance with a call-by-value reduction strategy.

By now, we are not able to reduce reducible *sub*expressions of function terms. To achieve this, we give a couple of congruence rules in the next definition. These also embody the yet missing part of the call-by-value reduction strategy, since we still need a way to fully evaluate the "right-hand side" of a reducible expression before the expression itself may be reduced.

Definition 3.11 (congruence rules).

A *congruence rule* for the evaluation of quantum closures has the form

$$\frac{\mathcal{C} \to_p \mathcal{D}}{\mathcal{C}' \to_p \mathcal{D}'} \text{ , where } \mathcal{C}, \mathcal{C}', \mathcal{D}, \mathcal{D}' \in \mathcal{S}_{QLC},$$

and is read as "whenever there is a probabilistic reduction rule of the form $\mathcal{C} \to_p \mathcal{D}$, then there is also a probabilistic reduction rule of the form $\mathcal{C}' \to_p \mathcal{D}'$."

For function terms in untyped QLC we define the following set of congruence rules (for all s, s', t, t', $t_1, t_2 \in \mathcal{T}_{uQLC}$ and $v, v_2 \in \mathcal{T}_{value}$):

$$\frac{[Q,\ L,\ t] \ \to_p\ [Q',\ L',\ t']}{[Q,\ L,\ s\ t] \ \to_p\ [Q',\ L',\ s\ t']} \qquad \frac{[Q,\ L,\ t] \ \to_p\ [Q',\ L',\ t']}{[Q,\ L,\ t\ v] \ \to_p\ [Q',\ L',\ t'\ v]}$$

$$\frac{[Q,\ L,\ t_2] \ \to_p\ [Q',\ L',\ t_2']}{[Q,\ L,\ \langle t_1,\ t_2\rangle] \ \to_p\ [Q',\ L',\ \langle t_1,\ t_2'\rangle]} \qquad \frac{[Q,\ L,\ t_1] \ \to_p\ [Q',\ L',\ t_1']}{[Q,\ L,\ \langle t_1,\ v_2\rangle] \ \to_p\ [Q',\ L',\ \langle t_1',\ v_2\rangle]}$$

$$\frac{[Q,\ L,\ t] \ \to_p\ [Q',\ L',\ t']}{[Q,\ L,\ inj_l(t)] \ \to_p\ [Q',\ L',\ inj_l(t')]} \qquad \frac{[Q,\ L,\ t] \ \to_p\ [Q',\ L',\ t']}{[Q,\ L,\ inj_r(t)] \ \to_p\ [Q',\ L',\ inj_r(t')]}$$

$$\frac{[Q,\ L,\ s] \ \to_p\ [Q',\ L',\ s']}{[Q,\ L,\ match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)] \ \to_p\ [Q',\ L',\ match\ s'\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)]}$$

As promised before, the first congruence rule enables the reduction of the right-hand side of a term application $(s\ t)$. Moreover, the second congruence rule allows the reduction of the left-hand side of a term application (for instance to a term abstraction, pair abstraction or term constant). From the third and fourth congruence rules, we immediately see that tuples in QLC are evaluated component-wise from right to left. Finally, the last three congruence rules allow the reduction of function terms t and s in disjoint unions $inj_l(t)$ and $inj_r(t)$ and in case distinctions ($match\ s\ with \ldots$), and furthermore, by recursive application of congruence rules, we may even reduce subexpressions inside of t and s.

Remark: Using our "less sugared" syntactic variant it is now crystal clear why there is only one congruence rule for case distinction terms $(match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r))$ (namely the one enabling reduction of function term s) and none for recursion terms $(letrec\ f = (\lambda x.s)\ in\ t)$. In both cases the included function terms $(\lambda x.t_l)$, $(\lambda y.t_r)$ and $(\lambda x.s)$ are already value terms by default, and thus no further reducible, anyway. Hence, congruence rules enabling their reduction would bring nothing new. The same also holds for the sugared variant in [SV09], however, there it is perhaps less obvious to come up with the explanation.

Moreover, we "lost" one congruence rule here, namely

$$\frac{[Q,\ L,\ t]\ \to_p\ [Q',\ L',\ t']}{[Q,\ L,\ let\ \langle x,\ y\rangle = t\ in\ s]\ \to_p\ [Q',\ L',\ let\ \langle x,\ y\rangle = t'\ in\ s]}\ .$$

On the one hand, this is not much of a surprise, since we have no syntactic construct as in the conclusion of this rule. On the other hand, when we consider Convention 1.3.2 in [SV09] and keep the intended computational behavior of the involved constructs in mind, we may come to the equality

$$let\ \langle x, y\rangle = t\ in\ s\ \equiv\ (\lambda\langle x, y\rangle.s)\ t\ .$$

And since the first congruence rule in Definition 3.11 applied to the right-hand side of $((\lambda\langle x, y\rangle.s)\ t)$ directly translates to

$$\frac{[Q,\ L,\ t]\ \to_p\ [Q',\ L',\ t']}{[Q,\ L,\ (\lambda\langle x, y\rangle.s)\ t]\ \to_p\ [Q',\ L',\ (\lambda\langle x, y\rangle.s)\ t']}\ ,$$

where we see we have the same possibilities with our "less sugared" variant and actually did not lose anything.

Definition 3.12 (evaluation of quantum closures).

An *evaluation step* $[Q, L, t] \to_p [Q', L', t']$ for a quantum closure $[Q, L, t]$ is the application of a probabilistic reduction rule $[Q, L, t] \to_p [Q', L', t']$ with the *evaluation result* $[Q', L', t']$.

An *evaluation* of a quantum closure $[Q, L, t]$ is a finite sequence of evaluation steps $[Q, L, t] \to_p^* [Q', L', t']$ with the *evaluation result* $[Q', L', t']$.

A *full evaluation* of a quantum closure $[Q, L, t]$ is an evaluation $[Q, L, t] \to_p^* [Q', L', t']$, where there is no evaluation step $[Q', L', t'] \to_p [Q'', L'', t'']$, i.e. no further reduction is possible. The evaluation result of a full evaluation is also called *final result*.

According to this definition, we evaluate an untyped *QLC* function term embedded into a quantum closure by applying the probabilistic reduction rules on quantum closures we defined in this section, respecting the given congruence rules. The goal of such successive reductions is to reach a state that is not reducible anymore, which means we either reached a value state or we got stuck with an so-called *error state* (see Definition 4.56 in section 4.4).

As we have already discussed along the road, we end up with a call-by-value evaluation strategy, due to the way we have defined reduction rules, congruence rules and evaluations. Operational semantics of untyped *QLC* hence does not reduce subexpressions inside the scope of term abstractions or pair abstractions as long as no argument is applied to these.

At the end of this section, we highlight one more characteristic of *QLC* function terms: there exist function terms t in untyped *QLC* for which a quantum closure $[Q, L, t]$ does not have a full evaluation, i.e. all possible evaluations do result in quantum closures that are not value states and are further reducible. One instance of such t is function term $letrec\ f = (\lambda x.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)$, as we can see by

$$\begin{aligned}
&\big[Q, L, letrec\ f = (\lambda x.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)\big] \\
&\qquad \to_1 \big[Q, L, (f\ \langle\rangle)[(\lambda x.(letrec\ f = (\lambda x.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)))\ /\ f]\big] \\
&\qquad \equiv\ \big[Q, L, (\lambda x.(letrec\ f = (\lambda x.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)))\ \langle\rangle\big] \\
&\qquad \to_1 \big[Q, L, letrec\ f = (\lambda x.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)\big]\ .
\end{aligned}$$

Also note, full evaluations for a particular quantum closure are not necessarily unique, and even worse, different full evaluations for one quantum closure may lead to disagreeing final results. (Disagreeing final results for disagreeing full evaluations will even become inevitable in the typed setting later, at least at the level of quantum closures. The function terms inside disagreeing final results, however, may be equal.) Consider again function term $\mathrm{coin} \equiv \big(\lambda z.\mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big)$ from Example 3.5. When we put it into an appropriate quantum closure and apply an argument to it, we obtain the following differing full evaluations for quantum closure $[\mathbb{1}, |\rangle, \mathrm{coin}\ \langle\rangle]$, although we use the same reduction strategy for both evaluations:

$$\begin{aligned}
\big[\mathbb{1}, |\rangle, \mathrm{coin}\ \langle\rangle\big] &\equiv \big[\mathbb{1}, |\rangle, \big(\lambda z.\mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big)\ \langle\rangle\big] \\
&\to_1 \big[\mathbb{1}, |\rangle, \mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big] \\
&\to_1 \big[|0\rangle, |q_1\rangle, \mathrm{meas}\ (H\ q_1)\big] \\
&\to_1 \Big[\tfrac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big), |q_1\rangle, \mathrm{meas}\ q_1\Big] \\
&\to_{\frac{1}{2}} \big[|0\rangle, |q_1\rangle, 0\big]
\end{aligned}$$

and

$$\begin{aligned}
\big[|\rangle, \mathrm{coin}\ \langle\rangle\big] &\equiv \big[\mathbb{1}, |\rangle, \big(\lambda z.\mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big)\ \langle\rangle\big] \\
&\to_1 \big[\mathbb{1}, |\rangle, \mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big] \\
&\to_1 \big[|0\rangle, |q_1\rangle, \mathrm{meas}\ (H\ q_1)\big] \\
&\to_1 \Big[\tfrac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big), |q_1\rangle, \mathrm{meas}\ q_1\Big] \\
&\to_{\frac{1}{2}} \big[|1\rangle, |q_1\rangle, 1\big]\ .
\end{aligned}$$

In this example the two different possibilities for a reduction of term application $(meas\ q_1)$ are obviously responsible for the disagreement of evaluations.

And since we have fixed the reduction strategy to call-by-value (and thus disagreeing evaluations as the informal ones in Example 3.5 cannot occur in the operational semantics of *QLC*), we may generalize the above observation: only probabilistic reduction steps which reduce (sub)expressions of the form $(meas\ q_j)$ can lead to disagreeing evaluations of quantum closures $[Q,\ |\ldots, q_j, \ldots\rangle,\ \ldots(\mathrm{meas}\ q_j)\ldots]$. Such reductions correspond to measurement of the qubit associated to q_j, and we have already said measurement processes in quantum physics are inherently random. Thus, disagreeing evaluations did not accidentally become part of *QLC*, but are rather a necessity for properly modeling computations in the quantum world.

4 Polymorphically typed Quantum Lambda Calculus

In the two previous sections we have outlined the most basic concepts and formalisms used in the field of quantum computation, on the one hand, and we have introduced a concrete calculus modeling classical computations as well as quantum computations, on the other hand. (By quantum computations, we actually mean quantum physical phenomenons and processes that can be interpreted as computations, to be more precise.) In section 2.2 we have presented a central theorem, namely Theorem 2.5 – widely known as the *no-cloning theorem* – which draws a strict line showing that cloning (i.e. exact copying without loss of information) of arbitrary and a-priori unknown quantum data is clearly beyond the possibilities approved by the laws of quantum physics.

An important question that we by now have neither posed nor answered is: what consequences does the no-cloning theorem entail for calculi such as QLC? Or, put into different words: what restrictions on the formulation of QLC function terms does this result impose if we want to be sure that the formulated function term describes computations that are actually physically realizable, at least in principle? This is a very interesting and quite important question indeed.

Untyped QLC, as we have presented it in section 3, is not subject to any restriction induced by Theorem 2.5. So we can without any difficulties formulate a function term $(\lambda f x.f\ x\ x)$, for instance. From the operational semantics' point of view this function term simply takes two arguments and produces a result where the second argument is applied twice, first to the first argument and afterwards to the result of this first application. The following example illustrates what is meant by this.

Example 4.1. Recall function term

$$\mathrm{xor} \equiv \big(\lambda xy.\mathit{if}\ x\ \mathit{then}\ (\mathit{if}\ y\ \mathit{then}\ 0\ \mathit{else}\ 1)\ \mathit{else}\ (\mathit{if}\ y\ \mathit{then}\ 1\ \mathit{else}\ 0)\big)$$

which we already encountered in Example 3.5. Let us now take a look at the computational behavior of function term $((\lambda f x.f\ x\ x)\ \mathrm{xor}\ 0)$ which we put into an appropriate quantum closure:

$$\begin{aligned}\left[\mathbb{1}, |\rangle, (\lambda f x.f\ x\ x)\ \mathrm{xor}\ 0\right] &\to_1 \left[\mathbb{1}, |\rangle, (\lambda x.\mathrm{xor}\ x\ x)\ 0\right] \\ &\to_1 \left[\mathbb{1}, |\rangle, \mathrm{xor}\ 0\ 0\right] \\ &\to_1^* \left[\mathbb{1}, |\rangle, 0\right]\end{aligned}$$

fully evaluates to a value state. Clearly, input term 0 is copied in the second evaluation step and thus contributes twice to the final result.

Copying of the second argument in the example evaluation is caused by the shape of function term $(\lambda f x.f\ x\ x)$. The reason is quite simple: term variable x occurs more than once in the scope of term abstraction $(\lambda x.f\ x\ x)$. Such multiple occurrences of term variables require copying during evaluation. This is not at all problematic as long as only classical information is concerned.

However, this is the point where Theorem 2.5 has a crucial impact on our considerations, since it entails copying is impossible for arbitrary and a-priori unknown quantum data. This means, although untyped QLC clearly allows the formulation of function terms such as

$$(\lambda x.f\ x\ x)\ (\mathrm{new}\ 0),$$

we may very well be in doubt whether the formulated function term does represent a computation which is actually physically realizable. Hence, the no-cloning theorem sheds new light on the way we have to deal with quantum data referenced in QLC function terms when we want to ensure we only formulate function terms whose evaluations will be in accordance with physical law.

Selinger and Valiron have drawn the following conclusion from this peculiarity of quantum computation: they define a *linear* type system for QLC in [SV09] that rules out multiple occurrences of function variables which link to quantum data. This in the end guarantees that well-typed function terms in *simply typed* QLC comply with the no-cloning theorem. In fact, the type system of simply typed QLC even gives stronger guarantees. Corollary 1.3.33 in [SV09] states that if a function term which is embedded in an appropriate quantum closure and which is equipped with an appropriate typing environment can be assigned a type in the simple type system, then an evaluation of this function term does not result in an error state.

For our discussion of the impact of the no-cloning theorem, we shall concentrate on *linearity* of simply typed QLC's type system. In usual algebra, we recognize a term as *linear in an indeterminate* x if x

occurs exactly once in the term. In analogy to this, we use the words "linear in x" also for function terms, however in a less strict sense. More precisely, we conceive a function term t as *linear in term variable* x, if x occurs *at most* once in t. Having this concept of linearity, we can now quite clearly state what it means to perform computations in *QLC* that do not break the no-cloning theorem. Given a (possibly infinite) sequence of evaluation steps

$$\begin{aligned} &\big[Q_0, L_0, t_0\big] \to_{p_1} \ldots \to_{p_n} \big[Q_n, L_n, t_n\big] \\ \text{or} \\ &\big[Q_0, L_0, t_0\big] \to_{p_1} \ldots \to_{p_n} \big[Q_n, L_n, t_n\big] \to_{p_{n+1}} \ldots \ , \end{aligned}$$

we conclude

> *t_0 models a computation in accordance with Theorem 2.5, if each of the t_i is linear in all $q_1, \ldots, q_{k_i}$ that appear in linking sequence $L_i = |q_1, \ldots, q_{k_i}\rangle$.*

However, such a check of *all* possible evaluations starting from an appropriate quantum closure $[Q_0, L_0, t_0]$ for function term t_0 is clearly impossible to do, especially if there is no full evaluation (which would be finite) for t_0. Thus, the type system needs to handle the situation differently. The solution is to assign types to function terms that allow us to individually keep track for each appearing term variable whether it is allowed to occur more than once or not. When we additionally recall the definition of quantum closures, where we require all free term variables in t_0 to appear in L_0, we can extend our check for accordance with physical law as follows:

> *t_0 models a computation in accordance with Theorem 2.5,*
>
> - *if t_0 is linear in each* free *term variable q, and*
> - *if for each term abstraction $(\lambda x{:}\Phi_x.s)$ in t_0 with* linear *type term Φ_x the scope s of the term abstraction is linear in x,*[21]
>
> *and an analogous condition holds for each pair abstraction and recursion term in t_0.*

Hence, one purpose of simply typed *QLC*'s type system is to guarantee the just stated conditions (in a slightly weakened variant[22]), and moreover, to make sure that all term variables referring to quantum data are assigned a linear type (namely a special type constant *Qbit*). Selinger and Valiron have successfully equipped *QLC* with such a simple type system, and thus a well-typed function term in simply typed *QLC* will not violate the no-cloning theorem during evaluation.

Of course, we want to keep this property when we extend simply typed *QLC*'s type system with parametric polymorphism in the current section. And in aiming at this, we face different challenges that we will discuss soon. However, let us first motivate the benefits of such a polymorphic extension a bit.

To illustrate the use of *QLC* and the limitations of its simple type system, we take a look at the following example function term:

$$t := \big\langle\ \ulcorner n \urcorner\ g\ (\lambda x.x)\ ,\ (\lambda\langle x, y\rangle.x)\ \big(\ulcorner n \urcorner\ (\lambda\langle x, y\rangle.\langle y, x\rangle)\ \langle \ulcorner 0 \urcorner, \ulcorner 1 \urcorner\rangle\big)\ \big\rangle\ ,$$

where

$$\ulcorner n \urcorner\ :\equiv\ \big(\lambda f x.\underbrace{f\ (\ldots(f}_{n \text{ times}}\ x)\ldots)\big)$$

represents a Church numeral (for $n \in \mathbb{N}$) and g shall stand for an arbitrary function (working on the identity function). We leave open what g does exactly, but clearly the second component of the pair computes the parity of n by switching places of numerals $\ulcorner 0 \urcorner$ and $\ulcorner 1 \urcorner$ in pair $\langle \ulcorner 0 \urcorner, \ulcorner 1 \urcorner\rangle$ n times followed by a projection to the first component.

When we look at function term t from the perspective of *simple* typing, i.e. without employing polymorphism of any kind, there is no reason why we should not be able to infer a proper type for each

[21] We formally define the mentioned syntactic constructs in section 4.3.

[22] The weakened variant allows one exception to the stated linearity conditions. The exception concerns case distinctions, where term variables of linear type may possibly occur once in each of the different branches $(\lambda x{:}\Phi_x.t_l)$ and $(\lambda y{:}\Phi_y.t_r)$, and thus, by nested occurrences of case distinctions, may in fact occur multiple times in function terms. We will investigate the details of this later, namely in subsection 4.6.3.

of the two occurrences of term $\ulcorner n \urcorner$, respectively. For instance, the following could do for the left-hand occurrence of $\ulcorner n \urcorner$

$$\Phi_{\text{left}} := \big((\Phi' \multimap \Phi') \multimap (\Phi' \multimap \Phi')\big) \multimap (\Phi' \multimap \Phi') \multimap (\Phi' \multimap \Phi') \ ,$$

for some type Φ', and for the right-hand occurrence we can find

$$\Phi_{\text{right}} := \big((\widehat{\Phi} \otimes \widehat{\Phi}) \multimap (\widehat{\Phi} \otimes \widehat{\Phi})\big) \multimap (\widehat{\Phi} \otimes \widehat{\Phi}) \multimap (\widehat{\Phi} \otimes \widehat{\Phi}) \ ,$$

where $\widehat{\Phi}$ stands as abbreviation of $((\Phi'' \multimap \Phi'') \multimap \Phi'' \multimap \Phi'')$ for some type Φ''. In the syntax we use in this section, "$\multimap$" denotes the function type operator and "$\otimes$" the operator for pair types (formal definitions will follow soon in section 4.1). Not at all surprising, we get the well known type pattern for Church numerals $(\Phi \multimap \Phi) \multimap \Phi \multimap \Phi$. So far, so good. But what happens, if we abstract from $\ulcorner n \urcorner$ and want it to be a parameter of the modeled function, say z? That means, we intend to use modified function term

$$\big(\lambda z.\langle\ z\ g\ (\lambda x.x)\ ,\ (\lambda\langle x, y\rangle.x)\ \big(z\ (\lambda\langle x, y\rangle.\langle y, x\rangle)\ \langle\ulcorner 0\urcorner, \ulcorner 1\urcorner\rangle\big)\ \rangle\big)$$

to gain more flexibility. At this point we already crossed the border of what the type system of simply typed *QLC* can handle properly. To infer a type for this term, we would need to unify types Φ_{left} and Φ_{right}, which is not possible for obvious reasons.

One possible (but certainly undesirable) solution is to give each occurrence of $\ulcorner n \urcorner$ its own abstraction and hand the same argument twice to the function, e.g. $(\lambda z_1 z_2.\langle \ldots \rangle)\ \ulcorner n \urcorner\ \ulcorner n \urcorner$. Now, this is a constrain that could not be enforced by the syntax or type system of *QLC*, but rather has the status of a convention among programmers. Moreover, it is clear that the intended semantics of this function term (almost certainly) changes and might be subject to abuse. (These last two objections need to be interpreted in the light of *QLC* having originally been developed with the intention in mind to create a practically usable programming language for quantum computation.)

Another possibility to fix the problem (and we will follow this suggestion in the current section) is to extend the type system in a convenient way and thus enable type inference for such constructions. Therefor we follow the well-known paths of parametric second-order polymorphism as it occurs in system *F*, for instance (see [Pie02], chapter 23, for an introduction focused on practical aspects of system *F*).

Our intended extension, however, comes at the price of some technical complications. These originate in the implicit linearity tracking Selinger and Valiron have chosen for their language. To keep the management of duplicable and linear terms away from the programmer, they have decided to omit syntactic constructs that make copying and discarding operations on linear elements and the removal of duplicability from types explicit. One consequence (the most obvious one for our work) are isomorphisms between types such as $!\Phi \cong !!\Phi$ and $!(\Phi \otimes \Psi) \cong (!\Phi \otimes !\Psi)$. Furthermore, in order to handle these isomorphisms, Selinger and Valiron introduce subtyping, which entails for example that a function of type $(!\Phi \multimap \Psi)$ accepts arguments of type $!\Phi$, of course, but also of type $!!\Phi$, which is not that surprising in the light of an isomorphism $!\Phi \cong !!\Phi$. However, the system goes even farther, since a function of type $(\Phi \multimap \Psi)$ also accepts arguments of type $!\Phi$. This is a desired and important property of subtyping (as for instance described in [Pie02], chapter 15), since a term of type $!\Phi$ is considered as a member of type Φ with the *additional* property of being duplicable.

As we have already said before, we want our polymorphic type system to still guarantee that term variables with a linear type may not occur twice in a well-typed term. Especially for term variables that are linked to quantum data we introduce a type constant *Qbit* for which we want to enforce linearity at all times. And since duplicability (nonlinearity, in other words) of a type is marked by exponentials, denoted "!", the introduction of polymorphism and the accompanying concept of type application (in analogy to term application in section 3) gives rise to the following problem: how can we prevent type substitutions (as result of a type application) of the form $(\forall X.!X)\ \mathit{Qbit} \rightsquigarrow (!X)[\mathit{Qbit}/X]$ yielding type $!\mathit{Qbit}$? Once again: $!\mathit{Qbit}$ shall be impossible to derive, since it could lead to computational behavior contradicting physical law, as we have already discussed above.

This leads to bounded quantification – a concept which results from combining universally quantified types with subtyping (see chapter 26 of [Pie02] for reference). In our example the right occurrence of X in $(\forall X.!X)$ may be seen as a "potentially dangerous spot", i.e. we need to prevent types that are supposed to be linear from being substituted for type variables such as X in this example. Therefore, we consider such occurrences of X *nonlinear* and allow only certain type terms to be substituted for type variables

that possess nonlinear occurrences. This is done by introducing an *upper bound* on the applicable types and that sort of *bounded quantification* is written as $(\forall X{<:}!\mathit{Top}.!X)$. We could interpret this as "X in $!X$ may be any type that is a subtype of $!\mathit{Top}$." Later on, we will see (Corollary 4.10) that any subtype of $!\mathit{Top}$ is of the form $!\Phi$ for some type term Φ. And because *Qbit* is not a subtype of $!\mathit{Top}$ and $!\mathit{Qbit}$ shall not be a derivable type in our system, we shall also not be able to end up with types $!\mathit{Qbit}$, $!!\mathit{Qbit}$ and the like by type applications. In this way, we reuse the subtyping concept that has been introduced into *QLC* by Selinger and Valiron to enable implicit linearity tracking and is thus available anyway, to now also enforce linearity of a certain class of type terms during evaluation of polymorphically typed *QLC* function terms. We postpone detailed formal considerations on this issue to subsection 4.2.3, because we first need to introduce the basic notions of our type system and to develop some formal tools to finally come back to this problem and investigate whether we have reached our goal.

In the subsequent sections we introduce type terms, the subtype relation, rules for deriving proved types and proved subtypes, and we study some properties that are inherent to our system. During the presentation we follow the pattern of definitions in the chapter on polymorphic functional type theory of Roy Crole's book [Cro93].[23]

4.1 Type terms

To get started, we introduce the raw material from which we build up our type system. This means we define the syntax of type terms and basic notions such as (nonlinear) free type variables, substitution of type variables and finally the subtype relation.

4.1.1 Basic definitions

Definition 4.2 (type terms).
Given a countably infinite set $\mathcal{V}_{type} := \{X, Y, Z, X_1, X_2 \dots\}$ of *type variables* (with $\mathcal{V}_{type} \cap \mathcal{V}_{term} = \emptyset$) and *type constants* *Top*, *Unit*, *Qbit*, we inductively define the set $\mathcal{T}_{type}$ of *type terms*:

- type constants:
  $\mathit{Top} \in \mathcal{T}_{type}$, $\mathit{Unit} \in \mathcal{T}_{type}$, $\mathit{Qbit} \in \mathcal{T}_{type}$,
- type variables:
  $X \in \mathcal{T}_{type}$ for all $X \in \mathcal{V}_{type}$,
- duplicable types:
  $!\Phi \in \mathcal{T}_{type}$, for all $\Phi \in \mathcal{T}_{type}$,
- composite types:
  $(\Phi \otimes \Psi) \in \mathcal{T}_{type}$, $(\Phi \oplus \Psi) \in \mathcal{T}_{type}$ and $(\Phi \multimap \Psi) \in \mathcal{T}_{type}$, for all $\Phi, \Psi \in \mathcal{T}_{type}$,
- bounded quantification:
  $(\forall X{<:}\Phi_X.\Psi) \in \mathcal{T}_{type}$, for all $X \in \mathcal{V}_{type}$ and for all $\Phi_X, \Psi \in \mathcal{T}_{type}$.

We call $(\Phi \otimes \Psi)$ a *product type*, $(\Phi \oplus \Psi)$ a *sum type*, $(\Phi \multimap \Psi)$ a *function type*, and $(\forall X{<:}\Phi_X.\Psi)$ *type abstraction*. Furthermore, the operator "!" is called *exponential*.[24,25] A type term $!\Phi$ with a leading exponential is called *duplicable* (or occasionally *nonlinear*) and a type term Φ without any leading exponential in called *linear*.

For type abstractions $(\forall X{<:}\Phi_X.\Psi)$, we call Ψ the *scope* of the type abstraction, and say that all *free* occurrences of X in Ψ are *bound*, and *captured by* $\forall X{<:}\Phi_X$. (The notion of "free occurrence" corresponds to the usual one, but we will soon define it formally.)

[23] From there we also lend some of the notation, such as function names *ftyv* (free type variables) and *ftmv* (free term variables), the style of using parenthesis (e.g. $(\forall X. \dots)$ instead of $\forall X(\dots)$), and again the way substitution deals with type abstractions to avoid unintended binding of free occurrences of X in substituted type term Υ by a type abstraction.

[24] The notation $\otimes$, $\oplus$, $\multimap$ and ! is borrowed from Jean-Yves Girard's *linear logic* in [Gir87], of which a fragment has been used by Selinger and Valiron as blueprint to develop their simple type system for *QLC*.

[25] The choice of the name "exponential" becomes clear when one thinks about "!" marking *nonlinear* elements (as in comparison to linear and nonlinear equations from algebra), i.e. elements that may contribute more then once to the overall result.

The three type constants *Top*, *Unit* and *Qbit* serve different purposes. *Top* stems from subtyping as it is presented in [Pie02], where it serves as supertype of all other types Φ, i.e. all types Φ are a subtype of *Top*. We will see this in the definition of the subtype relation in the next subsection. What also follows from the subtype relation is that we get a supertype of all duplicable types, as well, namely $!Top$ (see Corollary 4.10). This together with bounded quantification will later serve as the right tool to ensure that types such as $!Qbit$ cannot result from type application. Later we will discuss this in more detail.

The *Unit* type was already introduced in [SV09] as the singleton type (there denoted as "$\top$"); only function term $\langle\rangle$ – the *empty tuple* – (denoted "$*$" in [SV09]) is of type *Unit*, as we will see later.

Finally, *Qbit* is the type for the sort of data derived from quantum mechanical systems in quantum computation. It has also been introduced in [SV09]. In contrast to the other type constants, we shall enforce linearity for certain types (constant, composite and quantified) that involve *Qbit*. That is, we need to prohibit types such as $!Qbit$, $!(Qbit \otimes \Phi)$, $(\Phi \multimap !Qbit)$ and so on. We will repeatedly return to this discussion in the appropriate places. In a way, it is the major aim of this work to develop a type system that exhibits this property. Subsection 4.2.3 contains formal considerations on this issue.

The name "duplicable type" might be a bit misleading. The intended meaning is not that the type terms themselves are duplicable, but rather the elements which possess a duplicable type are duplicable (or reusable, in other words). Thus, a type term $(Qbit \otimes Qbit)$ (where *Qbit* is obviously not a duplicable type) does not pose any problem, although *Qbit* is used twice in this type term. Whereas a function term $\langle x, x\rangle$ of type $(Qbit \otimes Qbit)$ shall not be possible, because then x in each occurrence is of linear type *Qbit* but indeed occurs twice. Hence, we clearly face different challenges at the level of type terms than at the level of function terms. At the level of type terms, on the one hand, we need to take care that type constant *Qbit* never appears inside a duplicable type, i.e. type terms such as $!Qbit$, $!(Qbit \otimes Unit)$, $!(Qbit \oplus !Top)$, $(Unit \multimap !Qbit)$ shall be prohibited (please note that $!(Qbit \multimap Qbit)$ shall be allowed as we will discuss in more detail later). On the other hand, we need to prevent term variables that do not have a duplicable type from occurring more than once in a function term. The respective considerations on function terms will be treated in section 4.3.

Next, we recursively define the notion of *free type variables* in a type term:

Definition 4.3 (free type variables, α-equivalence).
Function $ftyv : \mathcal{T}_{type} \to \mathcal{P}(\mathcal{V}_{type})$ applied to a type term Φ yields the set $ftyv(\Phi)$ of all type variables that occur as *free type variables* in Φ, and is recursively defined as follows (for all $C \in \{Top, Unit, Qbit\}$):

$$\begin{aligned}
ftyv(C) &:= \emptyset \ , \\
ftyv(X) &:= \{X\} \ , \\
ftyv(!\Phi') &:= ftyv(\Phi') \ , \\
ftyv((\Phi' \otimes \Psi')) &:= ftyv(\Phi') \cup ftyv(\Psi') \ , \\
ftyv((\Phi' \oplus \Psi')) &:= ftyv(\Phi') \cup ftyv(\Psi') \ , \\
ftyv((\Phi' \multimap \Psi')) &:= ftyv(\Phi') \cup ftyv(\Psi') \ , \\
ftyv((\forall X{<:}\Phi_X.\Psi')) &:= ftyv(\Phi_X) \cup \big(ftyv(\Psi') \setminus \{X\}\big) \ .
\end{aligned}$$

With the notion of free and bound type variables in type terms, we can define α-equivalence on type terms: two type terms Φ and Ψ are considered *α-equivalent*, if they differ only in their bound type variables, i.e. if they have the same structure.

As usual, we identify type terms that are α-equivalent. From the definition of $ftyv((\forall X{<:}\Phi_X.\Psi'))$ we can see that the binding of X does not stretch over Φ_X (that is why we call solely Ψ' the scope of this type abstraction and not Φ_X and Ψ'). This means bounded quantification, as we use it here, does not produce recursive structures of any sort when we write type terms such as $(\forall X{<:}(X \otimes \Phi').\Psi')$, since X still occurs freely in $(X \otimes \Phi')$.

Definition 4.4 (substitution of type variables).
Substitution of free type variables in type terms is denoted as $\Phi[\Upsilon/Z]$, if all free occurrences of type variable Z in type term Φ are substituted by type term Υ, and is recursively defined as follows (for all $C \in \{Top, Unit, Qbit\}$):

$$
\begin{aligned}
C[\Upsilon/Z] &:= C\ , \\
Z[\Upsilon/Z] &:= \Upsilon\ , \\
X[\Upsilon/Z] &:= X\ , && \text{where } Z \neq X, \\
(!\Phi')[\Upsilon/Z] &:= \ !\Phi'[\Upsilon/Z]\ , \\
(\Phi' \otimes \Psi')[\Upsilon/Z] &:= (\Phi'[\Upsilon/Z] \otimes \Psi'[\Upsilon/Z])\ , \\
(\Phi' \oplus \Psi')[\Upsilon/Z] &:= (\Phi'[\Upsilon/Z] \oplus \Psi'[\Upsilon/Z])\ , \\
(\Phi' \multimap \Psi')[\Upsilon/Z] &:= (\Phi'[\Upsilon/Z] \multimap \Psi'[\Upsilon/Z])\ , \\
(\forall Z{<:}\Phi_Z.\Psi')[\Upsilon/Z] &:= (\forall Z{<:}\Phi_Z[\Upsilon/Z].\Psi')\ , \\
(\forall X{<:}\Phi_X.\Psi')[\Upsilon/Z] &:= (\forall X'{<:}\Phi_X[\Upsilon/Z].\Psi'[X'/X][\Upsilon/Z])\ , && \text{where } Z \neq X \text{ and} \\
& && X' \notin \mathit{ftyv}(\Psi') \cup \mathit{ftyv}(\Upsilon) \cup \{Z, X\}
\end{aligned}
$$

Let us take a closer look at the last two defining equations. Again, the scope Ψ' is ignored in the first one, since Z is bound in Ψ' by type abstraction. In the second line, we use renaming of bound type variables (inspired by [Cro93]) to avoid unintended binding of free variables. This is possible since we identify α-equivalent type terms.

Later on, we need to distinguish a special kind of free type variables which we call *nonlinear*. The idea is that substitutions of the form $(!X)[\Upsilon/X]$, $(!(X \otimes Y))[\Upsilon/X]$, $(!(Y \oplus X))[\Upsilon/X]$, $(!(\forall Y{<:}\Phi_Y.X))[\Upsilon/X]$ and similar ones bring type term Υ in a position, where it becomes part of a duplicable type term, even though it might itself be linear. However, the purpose of basing *QLC*'s type system on linear logic is to guarantee that data of the type *Qbit* is never marked as duplicable. Thus, in order to still provide this guarantee in the presence of second-order polymorphism, we need to recognize such occurrences of free type variables that turn substituted type terms into duplicable ones.

The following definition captures the idea of collecting free type variables which (when substituted) might produce duplicable type terms (in the way we just discussed) in at least one occurrence.

Definition 4.5 (nonlinear free type variables).
Function $\mathit{nftyv} : \mathcal{T}_{\mathit{type}} \to \mathcal{P}(\mathcal{V}_{\mathit{type}})$ collects all free type variables that occur as *nonlinear free type variables* in a type term, and is recursively defined as follows. For all $C \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\}$ and $n \geq 0$, we set

$$
\begin{aligned}
\mathit{nftyv}(!^n C) &:= \emptyset\ , \\
\mathit{nftyv}(X) &:= \emptyset\ , \\
\mathit{nftyv}(!^{n+1} X) &:= \{X\}\ , \\
\mathit{nftyv}(!^n(\Phi \otimes \Psi)) &:= \mathit{nftyv}(!^n\Phi) \cup \mathit{nftyv}(!^n\Psi)\ , \\
\mathit{nftyv}(!^n(\Phi \oplus \Psi)) &:= \mathit{nftyv}(!^n\Phi) \cup \mathit{nftyv}(!^n\Psi)\ , \\
\mathit{nftyv}(!^n(\Phi \multimap \Psi)) &:= \mathit{nftyv}(\Phi) \cup \mathit{nftyv}(\Psi)\ , \\
\mathit{nftyv}(!^n(\forall X{<:}\Phi_X.\Psi)) &:= \mathit{nftyv}(!^k\Phi_X) \cup \big(\mathit{nftyv}(!^n\Psi) \setminus \{X\}\big)\ , \\
& \qquad \text{where } k := \begin{cases} 1, \text{ if } X \in \mathit{nftyv}(!^n\Psi), \\ 0, \text{ otherwise.} \end{cases}
\end{aligned}
$$

An interesting aspect of this definition is how leading exponentials are propagated to recursive function calls on components of a type term and in which cases they are dropped and completely ignored.

Consider, for instance, the lines dealing with product and sum types. The type isomorphisms we mentioned earlier state that a product type is duplicable if and only if its components are, i.e. $!(\Phi \otimes \Psi) \cong (!\Phi \otimes !\Psi)$. Hence, leading exponentials are propagated to both components of the product and function *nftyv* is applied to the components with propagated exponentials. The same holds for sum types. (This will also be reflected in the rules for the derivation of proved types, since types $!(\Phi \otimes \Psi)$ and $!(\Phi \oplus \Psi)$ can only be derived if types $!\Phi$ and $!\Psi$ have been derived before.) On the other hand, the application of *nftyv* to function types with leading "!" signs drops all these leading exponentials completely in the recursive calls of *nftyv*. The fundamental difference between a product type $!(\Phi \otimes \Psi)$ or sum type $!(\Phi \oplus \Psi)$ and a function type $!(\Phi \multimap \Psi)$ leading to these distinctions becomes clear when we informally think about the abstract objects (i.e. function terms) having these types. An object $\langle t_1, t_2 \rangle$ of (duplicable) product type $!(\Phi \otimes \Psi)$ in a way *accommodates* abstract objects t_1 and t_2 (which then consequently need to be

duplicable as well), whereas an abstract object $(\lambda x.t)$ of (duplicable) function type $!(\Phi \multimap \Psi)$ *works on* an abstract object of type Φ and *results in* an object of type Ψ. Thus, abstract object $(\lambda x.t)$ which represents a function *working on* and *resulting in* objects of a certain type, may be duplicable no matter whether its type is composed of duplicable types or not.[26]

Finally, *nftyv* applied to a type abstraction propagates leading exponentials to the scope of the type abstraction. This is not surprising, for if we later eliminate the type abstraction by application of a type term, the type abstraction disappears but leading exponentials stay, e.g. function term $((\Lambda X{<:}\Phi_X.t)\ \Upsilon)$ is of type $!\Psi[\Upsilon/X]$ if function term $(\Lambda X{<:}\Phi_X.t)$ has type $!(\forall X{<:}\Phi_X.\Psi)$.

But, considering line $\mathit{nftyv}(!^n(\forall X{<:}\Phi_X.\Psi)) := \mathit{nftyv}(!^k\Phi_X) \cup \big(\mathit{nftyv}(!^n\Psi) \setminus \{X\}\big)$ again, where does the

$$k := \begin{cases} 1, \text{ if } X \in \mathit{nftyv}(!^n\Psi), \\ 0, \text{ otherwise} \end{cases}$$

originate from? In the right part of the union X is ignored (because it is bound by type abstraction). However, captured occurrences of X can still show nonlinear behavior when type application leads to substitution of bound variables. Hence, it is necessary to take captured nonlinear occurrences of X in $!^n\Psi$ into account when treating the type bound Φ_X for X.[27] Consider, for example, type term $!(\forall X{<:}Y.X)$. If we apply function *nftyv* to it, we get $\{Y\}$ as a result, due to the propagated exponential in $\mathit{nftyv}(!Y)$. This reflects (for instance) the case of a type variable Y being applied to a term of type $!(\forall X{<:}Y.X)$ resulting in a type $!Y$, where Y is clearly a nonlinear free type variable.

Now that we have established the basic notions of free type variables and their nonlinear versions, we formulate the following simple property which is already informally indicated by our choice of names and which will be of use later on.

Proposition 4.6. *Let Φ be an arbitrary type term. It holds $\mathit{nftyv}(\Phi) \subseteq \mathit{ftyv}(\Phi)$.*

Proof. By induction on the structure of Φ:

Base cases: Let $\Phi = C \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\}$. Then we have $\mathit{nftyv}(C) = \emptyset = \mathit{ftyv}(C)$. Let $\Phi = X \in \mathcal{V}_{\mathit{type}}$. Then $\mathit{nftyv}(X) = \emptyset \subset \{X\} = \mathit{ftyv}(X)$.

Induction cases: Let $\Phi = (\Phi' \otimes \Psi')$. Then, by induction, we have $\mathit{nftyv}(\Phi') \subseteq \mathit{ftyv}(\Phi')$ and $\mathit{nftyv}(\Psi') \subseteq \mathit{ftyv}(\Psi')$. And since $\mathit{nftyv}((\Phi' \otimes \Psi')) = \mathit{nftyv}(\Phi') \cup \mathit{nftyv}(\Psi')$ and $\mathit{ftyv}((\Phi' \otimes \Psi')) = \mathit{ftyv}(\Phi') \cup \mathit{ftyv}(\Psi')$, it immediately follows $\mathit{nftyv}((\Phi' \otimes \Psi')) \subseteq \mathit{ftyv}((\Phi' \otimes \Psi'))$, since $\cup$ is monotone with respect to $\subseteq$. The same holds for the cases of $\Phi = (\Phi' \oplus \Psi')$ and $\Phi = (\Phi' \multimap \Psi')$.

Let $\Phi = (\forall X{<:}\Phi_X.\Psi')$. Then, $\mathit{nftyv}((\forall X{<:}\Phi_X.\Psi')) = \mathit{nftyv}(!^k\Phi_X) \cup (\mathit{nftyv}(\Psi') \setminus \{X\})$ for

$$k = \begin{cases} 1, \text{if } X \in \mathit{nftyv}(\Psi'), \\ 0, \text{otherwise.} \end{cases}$$

Induction then tells us $\mathit{nftyv}(!^k\Phi_X) \subseteq \mathit{ftyv}(!^k\Phi_X) = \mathit{ftyv}(\Phi_X)$ and $\mathit{nftyv}(\Psi') \subseteq \mathit{ftyv}(\Psi')$. Together with monotonicity of $\cup$ with respect to $\subseteq$ and the fact that $\mathit{nftyv}(\Psi') \setminus \{X\} \subseteq \mathit{ftyv}(\Psi') \setminus \{X\}$ follows from $\mathit{nftyv}(\Psi') \subseteq \mathit{ftyv}(\Psi')$, this again leads to $\mathit{nftyv}((\forall X{<:}\Phi_X.\Psi')) \subseteq \mathit{ftyv}((\forall X{<:}\Phi_X.\Psi'))$.

Let $\Phi = !^{n+1}\Phi'$ for maximal n with $n \geq 0$, i.e. Φ' has no leading exponentials but Φ has at least one. We perform an induction on the structure of Φ':

Base cases: Let $\Phi' = C \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\}$. We get $\mathit{nftyv}(!^{n+1}C) = \emptyset = \mathit{ftyv}(!^{n+1}C)$.
Let $\Phi' = X \in \mathcal{V}_{\mathit{type}}$. Then we have $\mathit{nftyv}(!^{n+1}X) = \{X\} = \mathit{ftyv}(!^{n+1}X)$.

[26] What indeed matters for the typing of $(\lambda x.t)$, however, are free term variables possibly occurring in t. If these have a linear type then the whole function needs to get a linear type, as well. We will meet this fact again in section 4.3 on function terms.

[27] Please note that the isomorphism $!\Upsilon \cong !!\Upsilon$ for all $\Upsilon \in \mathcal{T}_{\mathit{type}}$ simplifies matters a lot at this point, because then it is sufficient to propagate at most one exponential to the recursive function call $\mathit{nftyv}(!^k\Phi_X)$. Without this isomorphism, we would have to isolate and propagate the exact number of exponentials that have an effect on the bound occurrences of X in $!^n(\forall X{<:}\Phi_X.\Psi)$, more precisely, the ones immediately captured by this type abstraction. It is not immediately clear how to do this, since there might be multiple (linear and nonlinear) occurrences of one type variable which are affected by different numbers of exponentials.

Induction cases: Let Φ' be a composite type or a type abstraction (without leading exponentials in both cases). We already covered similar situations in the other induction cases and hence we may conclude by slightly modified arguments that $\mathit{nftyv}(!^{n+1}\Phi') \subseteq \mathit{ftyv}(!^{n+1}\Phi')$.

□

4.1.2 The subtype relation $<:$

We have already said one or two words on our motivation for the employment of subtyping. It has been introduced into *QLC* by Selinger and Valiron to establish duplicability of types as an additional property, i.e. a term of duplicable type $!\Phi$ is, despite its duplicability, also of (possibly not duplicable) type Φ. And thus, in analogy to (naive) set theory, $!\Phi$ is called a *subtype* of Φ. How this notion extends to more complex types is captured in the definition of the subtype relation $<:$ (pronounced "sub").[28] In contrast to subtyping as it is presented, for instance, in [Pie02], simply typed *QLC* does originally not contain a type *Top* being supertype of all types. But since we intend to use subtyping to enforce duplicability of certain types (especially when it comes to substitution of nonlinear free type variables), we re-introduce *Top* into our polymorphic extension of simply typed *QLC*. To do so, we define the subtype relation in a way that *Top* is supertype of all types and that $!\mathit{Top}$ is supertype of all duplicable types but of none of the linear ones (see Corollary 4.10). Another interesting aspect of the supertype *Top* is that it enables simulation of *unbounded* quantification by type abstractions of the form $(\forall X{<:}\mathit{Top}.\Psi)$.

We now define the subtype relation $<: \subseteq \mathcal{T}_{\mathit{type}} \times \mathcal{T}_{\mathit{type}}$ as a preorder over $\mathcal{T}_{\mathit{type}}$, more precisely:

Definition 4.7 (subtype relation).
The *subtype relation* $<:$ is the smallest binary relation over $\mathcal{T}_{\mathit{type}}$ fulfilling the following axioms (for all $\Phi, \Psi \in \mathcal{T}_{\mathit{type}}$):

(1) $\Phi <: \Phi$,

(2) $\Phi <: \mathit{Top}$,

(3) $\Phi <: \Psi \implies !\Phi <: \Psi$,

(4) $!\Phi <: \Psi \implies !\Phi <: !\Psi$,

(5) $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies (\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)$,

(6) $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies (\Phi_1 \oplus \Phi_2) <: (\Psi_1 \oplus \Psi_2)$,

(7) $\Phi <: \Phi' \wedge \Psi <: \Psi' \implies (\Phi' \multimap \Psi) <: (\Phi \multimap \Psi')$,

(8) $\Phi_X <: \Phi'_X \wedge \Psi <: \Psi' \implies (\forall X{<:}\Phi'_X.\Psi) <: (\forall X{<:}\Phi_X.\Psi')$.

A statement $\Phi <: \Psi$ is read as "Φ is subtype of Ψ" or "Ψ is supertype of Φ", and can be interpreted as "whenever it is safe to use an abstract object (i.e. a function term) of type Ψ, it is also safe to use an object of type Φ." Using axioms (1) and (3), we get $!\Phi <: \Phi$ for all type terms Φ. This corresponds to our intuition (which we already came across several times) that function terms of type $!\Phi$ are also of type Φ with the additional property of being duplicable. Moreover, using axioms (1), (3) and (4), we get $!\Phi <: !!\Phi$ and $!!\Phi <: !\Phi$ which reflects the isomorphism $!\Phi \cong !!\Phi$.

Axiom (7) may seem a bit strange at first sight, since the direction of subtyping is reversed left of "$\multimap$" in contrast to the right-hand side. Pierce motivates this as follows:

> "The intuition is that if we have a function f of type $S_1 \to S_2$, then we know that f accepts elements of type S_1; clearly, f will also accept elements of any subtype T_1 of S_1. The type of f also tells us that it returns elements of type S_2; we can also view these results belonging to any supertype T_2 of S_2. That is, any function f of type $S_1 \to S_2$ can also be viewed as having type $T_1 \to T_2$.
>
> An alternative view is that it is safe to allow a function of one type $S_1 \to S_2$ to be used in a context where another type $T_1 \to T_2$ is expected as long as none of the arguments that may be passed to the function in this context will surprise it ($T_1 <: S_1$) and none of the results that it returns will surprise the context ($S_2 <: T_2$)." ([Pie02], page 185)

A similar intuition motivates the shape of axiom (8).

[28] The notation $<:$ is borrowed from type system $F_{<:}$, as it has been defined in [CMM91].

We have informally said above that we define $<:$ to be a preorder, but there is no axiom in its definition which explicitly introduces transitivity into our relation, although transitivity is a key property for preorders. This is true, of course. However, we can show that transitivity is a consequence of the axioms in Definition 4.7, as the following proposition states:

Proposition 4.8. *From* $\Phi <: \Upsilon$ *and* $\Upsilon <: \Psi$ *we may conclude* $\Phi <: \Psi$.

Proof. This is shown by induction on the structure of Υ.

Base case: Let $\Upsilon \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\} \cup \mathcal{V}_{type}$.
Then either $\Psi = \Upsilon$ (by axiom (1)) or $\Psi = \mathit{Top}$ (by axiom (2)). While we already have $\Phi <: \Psi = \Upsilon$ in the former case, we get $\Phi <: \Psi = \mathit{Top}$ by (2) in the latter one.

Induction cases: Let Υ be of the form $\Upsilon = (\Upsilon_1 \otimes \Upsilon_2)$.
Then $\Psi = \Upsilon$ (by (1)) or $\Psi = \mathit{Top}$ (by (2)) or Ψ is of the form $\Psi = (\Psi_1 \otimes \Psi_2)$ (by (5)). For the first two cases we get $\Phi <: \Psi$ as in the base case. In the third case, we know $\Upsilon_1 <: \Psi_1$ and $\Upsilon_2 <: \Psi_2$.

Regarding Φ, we have the following possibilities of its form: $\Phi = \Upsilon$ (by (1)) or $\Phi = !\Phi'$ (by (3)) or $\Phi = (\Phi_1 \otimes \Phi_2)$ (by (5)). The first case is again trivial. Let $\Phi = !\Phi'$. Then we need to distinguish three forms $\Phi' = \Upsilon$ or $\Phi' = !\Phi''$ or $\Phi' = (\Phi'_1 \otimes \Phi'_2)$, again. By now, it has become clear that we can iterate this case distinction arbitrarily often (which corresponds to an iterated application of axiom (3)). In this way, we end up with a structure for Φ of the form $\Phi = !^n\big(\Phi_1 \otimes \Phi_2\big)$ for a certain $n \geq 0$. Then we know $\Phi_1 <: \Upsilon_1$ and $\Phi_2 <: \Upsilon_2$ (which already covers the cases of $\Phi_1 = \Upsilon_1$ and $\Phi_2 = \Upsilon_2$ due to reflexivity of $<:$).

Now induction yields $\Phi_1 <: \Psi_1$ and $\Phi_2 <: \Psi_2$. Thus, we obtain (using axiom (5)) $(\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)$. To this we may then iteratively apply axiom (3) n times to eventually obtain $!^n(\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)$.

The case of Υ being of the form $(\Upsilon_1 \oplus \Upsilon_2)$ can be handled in an analogous way.

Let Υ be of the form $\Upsilon = (\Upsilon_1 \multimap \Upsilon_2)$. By an analogous inspection of the axioms of $<:$ as in the previous induction case, we get the following possible forms of Ψ and Φ:

- $\Psi = \mathit{Top}$ or $\Psi = (\Psi_1 \multimap \Psi_2)$ with $\Psi_1 <: \Upsilon_1$ and $\Upsilon_2 <: \Psi_2$, and
- $\Phi = !^n(\Phi_1 \multimap \Phi_2)$ with $\Upsilon_1 <: \Phi_1$ and $\Phi_2 <: \Upsilon_2$ and $n \geq 0$.

Hence, induction yields $\Psi_1 <: \Phi_1$ and $\Phi_2 <: \Psi_2$. Then, using axiom (7), we obtain $(\Phi_1 \multimap \Phi_2) <: (\Psi_1 \multimap \Psi_2)$. To this we may again iteratively apply axiom (3) n times to arrive at $!^n(\Phi_1 \multimap \Phi_2) <: (\Psi_1 \multimap \Psi_2)$.

The case of Υ being of the form $(\forall X{<:}\Upsilon_1.\Upsilon_2)$ is handled with a similar argument.

Let Υ be of the form $\Upsilon = !^{n+1}\Upsilon'$, $n \geq 0$, where Υ' shall be linear (of course, there exists exactly one such linear Υ' for each nonlinear Υ). Then we might again have trivial cases $\Psi = \Upsilon$ or $\Psi = \mathit{Top}$, which we treat as before. Furthermore, Ψ can be of the form $\Psi = !^k\Psi'$ for linear Ψ' with $\Upsilon' <: \Psi'$ and $k \geq 0$. This can be achieved starting from $\Upsilon' <: \Psi'$, then iteratively applying (3) $n+1$ times, followed by k applications of (4), and this way we end up with $!^{n+1}\Upsilon' <: !^k\Psi'$.[29]

Regarding the form of Φ, we end up with $\Phi = !^{l+1}\Phi'$ for linear Φ' with $\Phi' <: \Upsilon'$ and $l \geq 0$, which we obtain by similar reasoning as above, but where we have at least one leading exponential. This is because we cannot introduce leading exponentials in front of Υ' without having leading exponentials in front on Φ', when we start from linear Υ' and Φ'.

To eventually obtain $\Phi <: \Psi$, we start with $\Phi' <: \Psi'$ (which we get by induction), then apply (3) $l+1$ times, which yields at least one leading exponential in front of Φ' and afterwards apply (4) k times to in the end have $!^{l+1}\Phi' <: !^k\Psi'$ with $k, l \geq 0$.

□

[29] Of course, we could also have got $!^{n+1}\Upsilon' <: !^k\Psi'$ starting from nonlinear type terms $!^m\Upsilon$ and $!^m\Psi'$ with $\Upsilon' = \Psi'$ (getting $!^m\Upsilon <: !^m\Psi'$ by (1)), or from type terms $!^m\Upsilon'$ and $\Psi' = \mathit{Top}$ (getting $!^m\Upsilon' <: \mathit{Top}$ by (2)), where $m \leq \min(n+1, k)$ holds in both cases. But also here, we have $\Upsilon' <: \Psi'$, and thus the rest of our argument is still valid, in principle.

One may wonder why it is so interesting or even important for the subtype relation to be a preorder. We have emphasized the following intuition for a subtype statement $\Phi <: \Psi$ earlier: "whenever it is safe to use an abstract object of type Ψ, it is also safe to use an abstract object of type Φ." But then reflexivity and transitivity are clearly desirable characteristics for a subtype relation, since of course it shall be safe to replace an object of type Φ by another object of type Φ, on the one hand. And on the other hand, when we may replace an object of type Ψ by an object of type Υ and an object of type Υ by an object of type Φ without compromising safety in both cases, then we should also be able to replace an object of type Ψ directly by an object of type Φ and still be on the safe side. Hence, reflexivity and transitivity are quite natural properties which a subtype relation definitely should exhibit.

Figure 1 helps to get a better intuition of how the induced structure of the subtype relation looks like. In the figure we additionally find some counterexamples, which show that $<:$ is neither symmetric ($!\mathit{Top} <: \mathit{Top}$ and $\mathit{Top} \not<: !\mathit{Top}$) nor anti-symmetric ($!!\Phi <: !\Phi$ and $!\Phi <: !!\Phi$) nor total ($(!\Phi \otimes \Psi) \not<: (\Phi \otimes !\Psi)$ and $(\Phi \otimes !\Psi) \not<: (!\Phi \otimes \Psi)$).

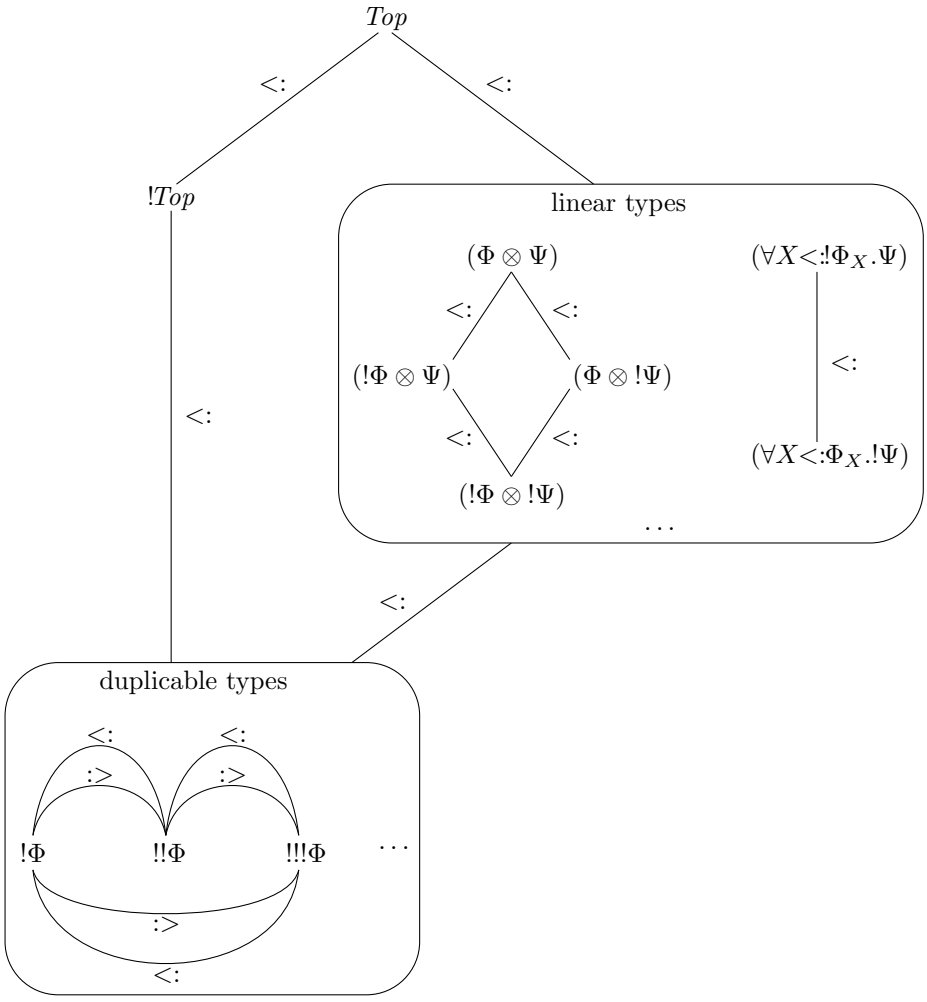


Figure 1: Structure of the subtype relation according to the axioms in Definition 4.7 and the results in Lemma 4.9 and Corollary 4.10. In the boxes labeled "linear types" and "duplicable types" the shown type terms are only interesting examples with selected relationships shown and are not to be understood as an exhaustive collection.

We next show two basic properties of the subtype relation which ensure that we followed the right route, so far. They reveal important facts on the structure of the subtype relation (in particular how linear and duplicable types are related), which we will use extensively later.

The following technically flavored lemma gives some insight into the structure of the subtype relation and lays the foundation for Corollary 4.10 in that it shows two things:

(i) duplicable types can only have duplicable subtypes, and

(ii) linear types never have a duplicable supertype.

Lemma 4.9 (adapted from [SV09])**.** *For all type terms $\Phi, \Psi \in \mathcal{T}_{type}$, it holds*

(i) If $\Phi <: !\Psi$, then $\Phi = !\Phi'$ for some type term $\Phi' \in \mathcal{T}_{type}$, and dually,

(ii) if $\Phi <: \Psi$ and Φ is not of the form $!\Phi'$, then Ψ is not of the form $!\Psi'$ for any type term $\Psi' \in \mathcal{T}_{type}$.

Proof.

(i): Assume $\Phi <: !\Psi$. Since $<:$ is defined as the smallest binary relation fulfilling axioms (1) to (8), and because there is no other axiom which may result in a leading exponential on the right-hand side of $<:$, $\Phi <: !\Psi$ can only be a result of axioms (1), (3) or (4). In all three of these cases, however, we must have a leading exponential on the left-hand side of $<:$, as well.

(ii): Let $\Phi <: \Psi$ and $\Phi \neq !\Phi'$ for each type term $\Phi' \in \mathcal{T}_{type}$. Then $\Phi <: \Psi$ can only be the result of one of the axioms (1), (2) and (5) to (8), since the other two axioms result in $\Phi = !\Phi'$. But then we immediately know that Ψ cannot have a leading exponential in case of axioms (5) to (8). In case of axiom (1), on the other hand, we get $\Psi = \Phi$, and we assumed Φ to be linear, i.e. $\Psi = \Phi \neq !\Phi'$ for any $\Phi' \in \mathcal{T}_{type}$.

□

The results in Corollary 4.10 ensure on the one hand that $!Top$ is supertype of all duplicable types, and that no linear type is subtype of $!Top$ on the other hand. It thus confirms that we can later use $!Top$ as upper type bound in an abstraction $(\forall X<:!Top.\Psi)$ to enforce that only duplicable types may be applied to a function term of this type.

Corollary 4.10. *For all type terms $\Phi \in \mathcal{T}_{type}$ it holds*

(i) $!\Phi <: !Top$, and

(ii) $\Phi \not<: !Top$, if $\Phi \neq !\Phi'$ for each $\Phi' \in \mathcal{T}_{type}$.

Proof. From axioms (2), (3) and (4) in the definition of the subtype relation it immediately follows that $!\Phi <: !Top$ for all type terms $\Phi \in \mathcal{T}_{type}$. On the other hand, it is impossible to have $\Phi <: !Top$ where $\Phi \neq !\Phi'$ for any type term $\Phi' \in \mathcal{T}_{type}$ due to the preceding Lemma 4.9. □

4.1.3 Connection with simply typed QLC's subtype relation

Before we continue to work with the subtype relation, let us first check whether we really formulated an extension of the subtype relation Selinger and Valiron use in [SV09]. Since our original intention was to keep the properties of simply typed *QLC* as far as possible while extending it with second order polymorphism, we shall make sure that we did not change central characteristics of the subtype relation more than we actually needed to. We do this by showing equivalence of axioms (1) and (3) to (7) to the axioms that are used in [SV09] (shown below, for reference). Please note we only omit axioms (2) and (8), which are exactly the ones that introduce the *Top* supertype and deal with bounded quantification. That means, they represent our intended extension (and nothing more).

Selinger and Valiron define their subtype relation as the smallest binary relation satisfying the following axioms (where $m, n \in \mathbb{N}$ and $(m = 0) \vee (n \geq 1)$ shall hold):

(i) $!^n Qbit <: !^m Qbit$,

(ii) $!^n Unit <: !^m Unit$,

(iii) $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies !^n(\Phi_1 \otimes \Phi_2) <: !^m(\Psi_1 \otimes \Psi_2)$,

(iv) $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies !^n(\Phi_1 \oplus \Phi_2) <: !^m(\Psi_1 \oplus \Psi_2)$,

(v) $\Phi <: \Phi' \wedge \Psi <: \Psi' \implies !^n(\Phi' \multimap \Psi) <: !^m(\Phi \multimap \Psi')$.

As said before, we consider the following reduced set of axioms defining our subtype relation for comparison:

(1) $\Phi <: \Phi$,

(3) $\Phi <: \Psi \implies !\Phi <: \Psi$,

(4) $!\Phi <: \Psi \implies !\Phi <: !\Psi$,

(5) $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies (\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)$,

(6) $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies (\Phi_1 \oplus \Phi_2) <: (\Psi_1 \oplus \Psi_2)$,

(7) $\Phi <: \Phi' \wedge \Psi <: \Psi' \implies (\Phi' \multimap \Psi) <: (\Phi \multimap \Psi')$.

According to Lemma 1.3.7 in [SV09] their subtype relation exhibits reflexivity and transitivity. Consequently, we may conclude that (i) to (v) imply (1). Furthermore, axioms (5), (6) and (7) are obviously special cases of axioms (iii), (iv) and (v) with $m = n = 0$, respectively.

Regarding axiom (3), we consider the derivation of an arbitrary subtype statement $\Phi <: \Psi$ from axioms (i) to (v). In the last derivation step the number of leading exponentials in Φ and Ψ are determined by n and m, respectively. But then, we can modify this last step through increasing the number n of exponentials for the type term left of $<:$ by one, thus yielding $!\Phi <: \Psi$ instead and respecting the required $(m = 0) \vee (n \geq 1)$. Hence, from (i) to (v), we may derive property (3).

Finally, property (4) results from axioms (i) to (v) by a very similar argument. However, in this case the leading exponential in front of Φ is necessary to not violate the requirement $(m = 0) \vee (n \geq 1)$.

Conversely, we now show that properties (i) to (v) result from axioms (1) and (3) to (7). As a first step, we derive the more restrictive properties:

(i') $\mathit{Qbit} <: \mathit{Qbit}$

(ii') $\mathit{Unit} <: \mathit{Unit}$

(iii') $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies (\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)$

(iv') $\Phi_1 <: \Psi_1 \wedge \Phi_2 <: \Psi_2 \implies (\Phi_1 \oplus \Phi_2) <: (\Psi_1 \oplus \Psi_2)$

(v') $\Phi <: \Phi' \wedge \Psi <: \Psi' \implies (\Phi' \multimap \Psi) <: (\Phi \multimap \Psi')$

While the first two properties (i') and (ii') are direct consequences of reflexivity axiom (1), properties (iii'), (iv') and (v') are exactly the same as axioms (5), (6) and (7). In a second step we derive an auxiliary property, namely

($\star$) $\Phi <: \Psi \implies !^n\Phi <: !^m\Psi$ for all $\Phi, \Psi \in \mathcal{T}_{type}$, where $(m = 0) \vee (n \geq 1)$,

which together with transitivity of the implication "$\Rightarrow$" helps us to confirm validity of axioms (i) to (v). For the derivation of ($\star$) we consider the following cases:

- $m = n = 0$:
  Trivially, we have $\Phi <: \Psi \implies !^0\Phi <: !^0\Psi$.
- $m = 0 \wedge n \geq 1$:
  By iterated application of axiom (3), we get $\Phi <: \Psi \Rightarrow !\Phi <: \Psi \Rightarrow \cdots \Rightarrow !^n\Phi <: \Psi$. Transitivity of implication then leads to $\Phi <: \Psi \implies !^n\Phi <: \Psi$.
- $m \geq 1 \wedge n \geq 1$:
  Using the previous case and iterated application of axiom (4), we get
  $\Phi <: \Psi \Rightarrow !^n\Phi <: \Psi \Rightarrow !^n\Phi <: !\Psi \Rightarrow \cdots \Rightarrow !^n\Phi <: !^m\Psi$. Transitivity of implication finally leads to $\Phi <: \Psi \implies !^n\Phi <: !^m\Psi$.
- $m \geq 1 \wedge n = 0$:
  This is not an allowed case. (At this point it becomes clear why axiom (4) requires a leading exponential in its premise and thus prevents relating type terms $!^n\Phi$ and $!^m\Psi$ as subtypes if they violate condition $(m = 0) \vee (n \geq 1)$.)

Hence, ($\star$) is a consequence of (3), (4) and transitivity of implication.

Now, having established properties (i') to (v') and ($\star$), we may use transitivity of implication to finally arrive at axioms (i) to (v).

Consequently, we may consider the subtype relation examined in the present work a true extension of Selinger's and Valiron's subtype relation for simply typed *QLC*. This concludes our detailed examination of the subtype relation we are using here and we can continue to define our polymorphic type system.

4.2 Proved types and proved subtypes

By now it has become clear that we do not allow all possible type terms to be used as types in our extended type system. To distinguish the special subset of type terms that we consider to be well-formed, we follow the usual tradition and define a couple of rules that enable us to derive proved types.

4.2.1 Basic definitions and properties

Before we can define proved types themselves, we need some more definitions.

Definition 4.11 (type contexts and related notions).

A *type context* Θ is a finite, possibly empty sequence of pairs (X, Φ_X) from set $\mathcal{V}_{type} \times \mathcal{T}_{type}$, written as $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$, where each type variable occurs at most once left of a $<:$ symbol.

$|\Theta|$ denotes the set of all type variables that occur left of a $<:$ symbol in type context Θ, e.g. $|X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n| = \{X_1, \ldots, X_n\}$. We will sometimes refer to this as the *domain* of Θ.

The concatenation of type contexts Θ and Θ' shall be denoted as juxtaposition Θ, Θ'. For such a concatenation to be consistent with the definition of a type context, we require $|\Theta| \cap |\Theta'| = \emptyset$ whenever we write Θ, Θ'.

Two type contexts Θ and Θ' are considered *α-equivalent* if $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$ and $\Theta' = X_1{<:}\Phi'_1, \ldots, X_n{<:}\Phi'_n$, and all corresponding type bounds in Θ and Θ' are α-equivalent, i.e. $\Phi_1 \equiv_\alpha \Phi'_1, \ldots, \Phi_n \equiv_\alpha \Phi'_n$, where $\equiv_\alpha$ denotes the α-equivalence relation (closed under reflexivity, symmetry and transitivity).

Consider an example where two type contexts Θ and Θ' are to be concatenated with a type context which contains only one pair (X, Φ_X). In this case we write $\Theta, \Theta', X{<:}\Phi_X$ for the concatenation and require three properties: $|\Theta| \cap |\Theta'| = \emptyset$ and additionally $X \notin |\Theta|$ and $X \notin |\Theta'|$. Please also note that $\Theta, \Theta', X{<:}\Phi_X$ and $\Theta, X{<:}\Phi_X, \Theta'$ represent different type contexts with $\Theta, \Theta', X{<:}\Phi_X \neq \Theta, X{<:}\Phi_X, \Theta'$, since we are dealing with (finite) sequences.

As we can see, a type context contains one type bound for each type variable that occurs left of a $<:$ in the context. This is necessary to keep track of bounds on the type of free type variables. If a type variable gets bound by a type abstraction, the type bound from the context will be taken over into the abstraction. But the bounds in type contexts also raise new questions and issues. The most obvious question is: when we have a type context $\Theta, X{<:}\Phi_X, \Theta'$ what is the scope of the definition of X? May we use free type variable X as part of the type bounds in Θ or Θ' or both? And may we use X also in its own type bound Φ_X? For a discussion of different approaches to scoping in the context of bounded quantification, see chapter 26 of Pierce's book [Pie02], pages 393–394. In our work we choose the easiest way of scoping, where X may only occur as free type variable in Θ', and make this precise in the next definition.

Definition 4.12 (well-scopedness of a type context).

A type context $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$ is considered *well-scoped*, if for all i with $1 \leq i \leq n$ and for all $Y \in \mathit{ftyv}(\Phi_i)$ we have $Y \in \{X_1, \ldots, X_{i-1}\}$.

This will avoid circular and recursive constructions such as $\Theta = Y{<:}Z, Z{<:}Y$ or $\Theta = X{<:}(X \oplus X)$.

Although the decision for a particular scoping strategy constitutes an important design choice, our notion of well-scoped type contexts is too weak to be sufficient for our purposes. Hence, we define the stronger notion of *consistency* for type contexts that will play a central role in the definition of derivation rules for proved types and proved subtypes as well as in some lemmas and propositions that we will provide. (Actually, the check for consistency of certain type contexts is what makes most of the upcoming proofs a bit tedious.) We will see soon (in Corollary 4.21) that well-scopedness of type contexts is indeed a weak version of the notion of consistency which we define now.

Definition 4.13 (consistency of a type context).
We call a type context $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$ *consistent*, denoted $\vdash \Theta$, if for all i with $1 \leq i \leq n$ we can derive $X_1{<:}\Phi_1, \ldots, X_{i-1}{<:}\Phi_{i-1} \vdash \Phi_i$ as a proved type according to the rules given below; in particular, $\vdash \Phi_1$ needs to be a proved type, where the type context left of the turnstile symbol is empty. The empty type context is considered to be consistent, which we denote by $\vdash \emptyset$.

As a next step, we need to define the sort of syntactical expressions that we use to denote proved types and proved subtypes. The syntax of these expressions is also taken over from chapter 5 of Roy Crole's book [Cro93].

Definition 4.14 (judgement, type-in-context, subtype-in-context).
We call syntactical expressions of the form $\Theta \vdash \Phi$ and similar ones *judgements*. A judgement $\Theta \vdash \Phi$ with a type context Θ and a type term Φ shall be called *type-in-context*, and a judgement of the form $\Theta \vdash \Phi <: \Psi$, where Ψ is a type term as well, is called *subtype-in-context*.

Now we arrived at the heart of our polymorphic extension. That means we now present the way of how we derive well-formed type terms. In the following definition we give a couple of rules that enable us to derive proved types and also proved subtypes. For presentation we follow the usual convention that necessary premises are (sub)types-in-context (or more general conditions) written above a horizontal line (sometimes stacked one over another to save space) and the derived (sub)type-in-context is written as conclusion beneath the line. We label each rule with a name to the right of the horizontal line.

Definition 4.15 (proved types and proved subtypes).
A *proved type* (*proved subtype*) is a type-in-context (subtype-in-context) that can be derived using the following rules, where $n \geq 0$.

Proved types:

$$\frac{\vdash \Theta}{\Theta \vdash !^n \mathit{Top}} \text{ (}\mathit{Top}\text{ type)} \qquad \frac{\vdash \Theta}{\Theta \vdash !^n \mathit{Unit}} \text{ (}\mathit{Unit}\text{ type)} \qquad \frac{\vdash \Theta}{\Theta \vdash \mathit{Qbit}} \text{ (}\mathit{Qbit}\text{ type)}$$

$$\frac{\vdash \Theta, X{<:}\Phi_X, \Theta'}{\Theta, X{<:}\Phi_X, \Theta' \vdash X} \text{ (linear type variable)} \qquad \frac{\Theta \vdash \Phi \qquad \Theta \vdash \Psi}{\Theta \vdash !^n(\Phi \multimap \Psi)} \text{ (function type)}$$

$$\frac{\vdash \Theta, X{<:}!\Phi_X, \Theta'}{\Theta, X{<:}!\Phi_X, \Theta' \vdash !^n X} \text{ (nonlinear type variable)}$$

$$\frac{\Theta \vdash !^n\Phi \qquad \Theta \vdash !^n\Psi}{\Theta \vdash !^n(\Phi \otimes \Psi)} \text{ (product type)} \qquad \frac{\Theta \vdash !^n\Phi \qquad \Theta \vdash !^n\Psi}{\Theta \vdash !^n(\Phi \oplus \Psi)} \text{ (sum type)}$$

$$\frac{\vdash \Theta, \Theta' \qquad \Theta, X{<:}\Phi_X, \Theta' \vdash \Psi \qquad X \notin \mathit{nftyv}(\Psi)}{\Theta, \Theta' \vdash (\forall X{<:}\Phi_X.\Psi)} \text{ (linear-polymorphic type)}$$

$$\frac{\Theta, \Theta' \vdash \Phi_X <: !\mathit{Top} \qquad \Theta, X{<:}\Phi_X, \Theta' \vdash !^n\Psi}{\Theta, \Theta' \vdash !^n(\forall X{<:}\Phi_X.\Psi)} \text{ (nonlinear-polymorphic type)}$$

Proved subtypes:

$$\frac{\Theta \vdash \Phi}{\Theta \vdash \Phi <: \mathit{Top}} \text{ (}\mathit{Top}\text{ supertype)} \qquad \frac{\Theta \vdash \Phi}{\Theta \vdash \Phi <: \Phi} \text{ (<: reflexivity)}$$

$$\frac{\Theta \vdash \Phi <: \Psi}{\Theta \vdash !\Phi <: \Psi} \text{ (! left)} \qquad \frac{\Theta \vdash !\Phi <: \Psi}{\Theta \vdash !\Phi <: !\Psi} \text{ (! right)}$$

$$\frac{\Theta \vdash \Phi <: \Phi' \qquad \Theta \vdash \Psi <: \Psi'}{\Theta \vdash (\Phi' \multimap \Psi) <: (\Phi \multimap \Psi')} \text{ (function subtype)}$$

$$\frac{\Theta \vdash \Phi_1 <: \Psi_1 \qquad \Theta \vdash \Phi_2 <: \Psi_2}{\Theta \vdash (\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)} \text{ (product subtype)}$$

$$\frac{\Theta \vdash \Phi_1 <: \Psi_1 \qquad \Theta \vdash \Phi_2 <: \Psi_2}{\Theta \vdash (\Phi_1 \oplus \Phi_2) <: (\Psi_1 \oplus \Psi_2)} \text{ (sum subtype)}$$

$$\frac{\Theta, \Theta' \vdash \Phi_X <: \Phi'_X \qquad \Theta, X{<:}\Phi_X, \Theta' \vdash \Psi <: \Psi'}{\Theta, \Theta' \vdash (\forall X{<:}\Phi'_X.\Psi) <: (\forall X{<:}\Phi_X.\Psi')} \text{ (polymorphic subtype)}$$

Before we discuss the above derivation rules in detail, we first extend the notion of α-equivalence to proved types and subtypes:

- two proved types $\Theta \vdash \Phi$ and $\Theta' \vdash \Phi'$ are called *α-equivalent* if type contexts Θ, Θ' are α-equivalent and type terms Φ, Φ' are α-equivalent, and
- two proved subtypes $\Theta \vdash \Phi <: \Psi$ and $\Theta' \vdash \Phi' <: \Psi'$ are considered α-equivalent, if all the Θ, Θ' and Φ, Φ' and Ψ, Ψ' are α-equivalent, respectively.

Henceforth, we identify α-equivalent proved types and proved subtypes.

Let us now take a closer look at some of the derivation rules. The first three rules (*Top* type), (*Unit* type) and (*Qbit* type) show the characteristics of axioms, which becomes immediately obvious when type contexts Θ are empty. Here we also see the special status of type constant *Qbit* among the other type constants, since it is not allowed to have any leading exponentials.

Rules (linear type variable) and (nonlinear type variable) can *only almost* be counted as axioms, since each of them requires a nonempty type context, and is thus based on at least one proved type $\vdash \Phi_X$. The two rules complement each other, although there is a certain overlap in their possible application. Proved type $\Theta, X{<:}!\Phi'_X, \Theta' \vdash X$ can be derived either by

$$\frac{\vdash \Theta, X{<:}\Phi_X, \Theta'}{\Theta, X{<:}\Phi_X, \Theta' \vdash X} \text{ (linear type variable)}$$

where we set $\Phi_X = !\Phi'_X$ or by

$$\frac{\vdash \Theta, X{<:}!\Phi'_X, \Theta'}{\Theta, X{<:}!\Phi'_X, \Theta' \vdash !^0 X} \text{ (nonlinear type variable)}$$

since the number of leading exponentials in front of X is allowed to be zero. Thus, both rules also allow less leading exponentials in the type term right of $\vdash$ than the type bound of X in the type context has. This reflects "one direction" of the isomorphism $!\Phi \cong !!\Phi$ (read from right to left). The other direction is embodied by $!^n$ in the conclusion of rule (nonlinear type variable), thus allowing more leading exponentials to appear in front of X (right of $\vdash$) than there are in the type bound for X in the respective type context. However, by requiring at least one leading exponential in the corresponding type bound, we prevent proved types such as $X{<:}\mathit{Qbit} \vdash !^n X$, which do not fit our intentions and would be possible using a hypothetical derivation rule

$$\frac{\vdash \Theta, X{<:}\Phi_X, \Theta'}{\Theta, X{<:}\Phi_X, \Theta' \vdash !^n X} \ .$$

Among the rules for derivation of composite types, rules (product type) and (sum type) can propagate leading exponentials from their premises to the respective conclusion. Put differently, it is only possible to derive a product or sum type with n leading exponentials if both of its components can be derived having (at least) n leading exponentials. Consequently, it is not possible to derive a duplicable product or sum type from types that are not duplicable. This is in contrast to the rule for derivation of function types, which matches the interpretation of duplicable functions we already mentioned in the discussion of function *nftyv*.

Regarding rules (linear-polymorphic type) and (nonlinear-polymorphic type) we notice a strong similarity between them. We could make them even more similar if we replaced the first premise of the former rule with $\Theta, \Theta' \vdash \Phi <: \mathit{Top}$. We will soon show that we would not lose any information through this replacement, since $\Theta, \Theta' \vdash \Phi <: \mathit{Top}$ entails consistency of Θ, Θ' (see Lemma 4.19). However, this "stronger" premise is not necessary, because the additional information that Φ is a subtype of *Top* is nothing new, since every (well-formed) type is a subtype of *Top* as rule (*Top* supertype) explicitly states. Hence, the difference between the linear and nonlinear version, on the one hand, is that the linear one requires the type variable which is to be bound not to appear as nonlinear free type variable in the scope of the derived abstraction. On the other hand, only the nonlinear version of the rule for polymorphic types allows leading exponentials in front of Ψ to be propagated to the derived type abstraction and the price we have to pay for this is that the maximally possible (w.r.t. $<:$) bound for X gets lowered to the supertype of all duplicable types. This shall ensure that in future type applications only duplicable types are substituted for free type variable X that may have nonlinear occurrences in Ψ. As in the case of the linear and nonlinear rules introducing type variables, the two rules for type abstractions complement each other, and they exhibit a certain overlap in their possible application, as well.

Moreover, the consistency check in the first premise of rule (linear-polymorphic type) makes sure that free type variable X and the associated type bound Φ_X cannot be used outside the scope of type abstraction ($\forall X{<:}\Phi_X.\Psi$). In rule (nonlinear-polymorphic type), however, this explicit check is not necessary, since it is already checked implicitly in the derivation of the first premise of that rule (again, see Lemma 4.19).

In rule (nonlinear-polymorphic type) the first premise might seem too large (i.e. we might be tempted to remove Θ' left of $\vdash$), since the second premise (more precisely, the consistency of its context) already ensures that $\Theta \vdash \Phi$ is a proved type, and thus we could derive $\Theta \vdash \Phi <: !\mathit{Top}$ without the need for any information from Θ'. However, the first premise, as it is, ensures consistency of type context Θ, Θ', and thus guarantees X does not appear freely in Θ', as mentioned above.

The derivation rules for proved subtypes resemble the axioms of the subtype relation, but now need type contexts to work, and thus subtype derivations can only start from proved types. As a consequence, we cannot derive a proved subtype $\Theta \vdash \Phi <: \Psi$ for arbitrary type terms Φ, Ψ, even if $\Phi <: \Psi$ holds. A quite simple example is type term $(!\mathit{Qbit} \otimes !\mathit{Qbit})$, for which we cannot derive proved type $\Theta \vdash (!\mathit{Qbit} \otimes !\mathit{Qbit})$ for any type context Θ, as we will see later in subsection 4.2.3. With respect to the subtype relation, we clearly have $(!\mathit{Qbit} \otimes !\mathit{Qbit}) <: \mathit{Top}$ by axiom (1). However, since we cannot derive $\Theta \vdash (!\mathit{Qbit} \otimes !\mathit{Qbit})$, we can also not apply rule (*Top* supertype) to it, and thus we do not have a way to obtain proved subtype $\Theta \vdash (!\mathit{Qbit} \otimes !\mathit{Qbit}) <: \mathit{Top}$ (the other derivation rules are not helpful in this case).

On the other hand, the derivation rules for proved subtypes are not strict enough to make sure that type terms appearing in a proved subtype do in general have a proved type as counterpart. Let us look at an example for this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash \mathit{Qbit}}\,(\mathit{Qbit}\text{ type})}{\vdash \mathit{Qbit} <: \mathit{Qbit}}\,(<:\text{ reflexivity})}{\vdash !\mathit{Qbit} <: \mathit{Qbit}}\,(!\text{ left})}{\vdash !\mathit{Qbit} <: !\mathit{Qbit}}\,(!\text{ right})$$

is a proper derivation of proved subtype $\vdash !\mathit{Qbit} <: !\mathit{Qbit}$. However, $\vdash !\mathit{Qbit}$ cannot be derived (and shall not be derivable), as we will see in subsection 4.2.3.

Let us now turn our attention to basic aspects of proved subtypes which indeed allow us to draw conclusions about the type terms that occur in the derived subtyping statement. At first, we notice that whenever we can derive a proved subtype $\Theta \vdash \Phi <: \Psi$, then we know Φ is related to Ψ with respect to the subtype relation.

Proposition 4.16. *If $\Theta \vdash \Phi <: \Psi$ is a proved subtype, then $\Phi <: \Psi$.*

Proof. By straightforward induction on the derivation of $\Theta \vdash \Phi <: \Psi$. □

Hence, we may immediately lift Lemma 4.9 and Corollary 4.10 to the case of proved subtypes:

Corollary 4.17. *For all type terms $\Phi, \Psi \in \mathcal{T}_{type}$ it holds*

- *if $\Theta \vdash \Phi <: !\Psi$ is a proved subtype, then $\Phi = !\Phi'$, for some type term $\Phi' \in \mathcal{T}_{type}$, and dually,*
- *if $\Theta \vdash \Phi <: \Psi$ is a proved subtype and Φ is not of the form $!\Phi'$, then Ψ is not of the form $!\Psi'$ for any type term $\Psi' \in \mathcal{T}_{type}$.*

In particular, we have for all proved types $\Theta \vdash \Phi$

- *$\Theta \vdash !\Phi <: !Top$ is a proved subtype, whereas*
- *$\Theta \vdash \Phi <: !Top$ cannot be derived, if $\Phi \neq !\Phi'$ for all $\Phi' \in \mathcal{T}_{type}$.*

Proof. There are only three rules by which $\Theta \vdash \Phi <: !\Psi$ could have been derived:

(<: reflexivity). Then Φ must have the form $\Phi = !\Psi$.

(! left). Then we immediately know Φ is of the form $\Phi = !\Phi'$.

(! right). The only premise of this rule states $\Theta \vdash !\Phi' <: \Psi$, i.e. Φ must be of the form $\Phi = !\Phi'$.

All other derivation rules for proved subtypes do not allow leading exponentials right of <: in their conclusions. This proves the first two propositions in Corollary 4.17 and the last of its items, as well, since it is a special case of the second proposition. The third item of this corollary, however, is valid since we can perform the following derivation for each proved type $\Theta \vdash \Phi$:

$$\dfrac{\dfrac{\dfrac{\Theta \vdash \Phi}{\Theta \vdash \Phi <: Top}\,(Top \text{ supertype})}{\Theta \vdash !\Phi <: Top}\,(!\text{ left})}{\Theta \vdash !\Phi <: !Top}\,(!\text{ right})$$

□

For the subtype relation we have shown in Proposition 4.8 that <: is transitive. In general, the case is somewhat different for proved subtypes. Consider again type term $(!Qbit \otimes !Qbit)$. We can perform subtype derivations

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash Qbit}\,(Qbit \text{ type})}{\vdash Qbit <: Qbit}\,(<: \text{ reflexivity})}{\vdash !Qbit <: Qbit}\,(!\text{ left})}{\vdash (!Qbit \otimes !Qbit) <: (Qbit \otimes Qbit)}\,(\text{product subtype}) \qquad \dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash Qbit}\,(Qbit \text{ type})}{\vdash Qbit <: Qbit}\,(<: \text{ reflexivity})}{\vdash !Qbit <: Qbit}\,(!\text{ left})$$

and

$$\dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash Qbit}\,(Qbit \text{ type}) \qquad \dfrac{\vdash \emptyset}{\vdash Qbit}\,(Qbit \text{ type})}{\vdash (Qbit \otimes Qbit)}\,(\text{product type})}{\vdash (Qbit \otimes Qbit) <: Top}\,(Top \text{ supertype})\,.$$

However, we have already mentioned above that $\Theta \vdash (!Qbit \otimes !Qbit) <: Top$ is not derivable for any type context Θ. Hence, we do not have transitivity for proved subtypes in general. Nevertheless, we will take a much deeper look into this issue in subsection 4.2.4. But we first need a better insight into our system and to develop the necessary tools before we can eventually show in Theorem 4.47 that we have at least a weak form of transitivity for proved subtypes. More precisely, we can derive a proved subtype $\Theta \vdash \Phi <: \Psi$ if $\Theta \vdash \Phi$ and $\Theta \vdash \Psi$ are proved types which are transitively connected by a chain of proved subtypes $\Theta_1 \vdash \Phi <: \Upsilon_1$, $\Theta_2 \vdash \Upsilon_1 <: \Upsilon_2$, …, $\Theta_n \vdash \Upsilon_{n-1} <: \Psi$.

Now that we have identified the types we are interested in for our further investigations and know how to derive them, we can examine some of the basic properties that our derivation rules entail for proved types and proved subtypes.

We start by showing that our derivation rules preserve consistency of type contexts. This is a fact we use extensively in the upcoming proofs. It again shows that consistency for type contexts is a key notion in our system.

Lemma 4.18. *If $\Theta \vdash \Phi$ is a proved type, then Θ is consistent.*

Proof. We prove this by induction on the derivation of $\Theta \vdash \Phi$.

Base cases: Suppose $\Theta \vdash !^n \mathit{Top}$ has been derived using rule (*Top* type). The respective premise states Θ is consistent. And since Θ occurs unchanged in the derived conclusion, it is trivially consistent in the derived proved type $\Theta \vdash !^n \mathit{Top}$. The same argument holds for rules (*Unit* type), (*Qbit* type), (linear type variable) and (nonlinear type variable).

At first glance, the case of rule (linear-polymorphic type) looks more complicated, but it can be handled in the same manner, because the first premise states consistency of Θ, which is of the form Ξ, Ξ' in this case. But since the conclusion is proved type $\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi)$, its type context $\Theta = \Xi, \Xi'$ thus is consistent.

Induction cases: In rules (function type), (product type) and (sum type) type context Θ occurs unchanged in the proved types of the premises, and thus Θ in the respectively derived types $\Theta \vdash \Phi$ is consistent by induction.

Rule (nonlinear-polymorphic type) is somewhat more complicated, since the first premise contains a proved subtype $\Xi, \Xi' \vdash \Phi_X <: !\mathit{Top}$. We first notice that Corollary 4.17 implies Φ_X must be of the form $!\Phi'_X$ for some $\Phi'_X \in \mathcal{T}_{type}$. Taking a look at the rules for deriving proved subtypes, we then see there are essentially two ways of how to derive $\Xi, \Xi' \vdash !\Phi'_X <: !\mathit{Top}$. One possible derivation is

$$\dfrac{\dfrac{\dfrac{\dfrac{\Xi, \Xi' \vdash !^k \Phi''_X}{\Xi, \Xi' \vdash !^k \Phi''_X <: \mathit{Top}}\,(\mathit{Top}\text{ supertype})}{\Xi, \Xi' \vdash !^{k+1} \Phi''_X <: \mathit{Top}}\,(!\text{ left})}{\vdots}\,(!\text{ left})}{\Xi, \Xi' \vdash !^{n+1} \Phi''_X <: !\mathit{Top}}\,(!\text{ right})$$

where we start from a type term Φ''_X (with $\Phi_X = !^{n+1}\Phi''_X$ for some $n \geq k \geq 0$) and perform an appropriate number of (! left) steps, possibly none at all. The (! right) step might have been taken earlier, but we have to take at least one. The other possible derivation has the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\Xi, \Xi' \vdash \mathit{Top}}{\Xi, \Xi' \vdash \mathit{Top} <: \mathit{Top}}\,(<:\text{ reflexivity})}{\Xi, \Xi' \vdash !^{n+1} \mathit{Top} <: \mathit{Top}}\,(!\text{ left})}{\vdots}\,(!\text{ left})}{\Xi, \Xi' \vdash !^{n+1} \mathit{Top} <: !\mathit{Top}}\,(!\text{ right})\,,$$

where again rule (! right) might have been applied earlier. A slightly different variant of the latter derivation is

$$\dfrac{\dfrac{\dfrac{\dfrac{\Xi, \Xi' \vdash !\mathit{Top}}{\Xi, \Xi' \vdash !\mathit{Top} <: !\mathit{Top}}\,(<:\text{ reflexivity})}{\Xi, \Xi' \vdash !^{2} \mathit{Top} <: !\mathit{Top}}\,(!\text{ left})}{\vdots}\,(!\text{ left})}{\Xi, \Xi' \vdash !^{n+1} \mathit{Top} <: !\mathit{Top}}\,(!\text{ left})\,.$$

In the last two variants we clearly have $\Phi_X = !^{n+1}\mathit{Top}$ for some $n \geq 0$. By induction, we know Ξ, Ξ' is consistent in all these cases, and from the above derivations we can see $\Theta = \Xi, \Xi'$ remains untouched in each step. □

The same property holds for proved subtypes:

Lemma 4.19. *If $\Theta \vdash \Phi <: \Psi$ is a proved subtype, then Θ is consistent.*

Proof. We prove this by induction on the derivation of $\Theta \vdash \Phi <: \Psi$.

Base cases: In rules (*Top* supertype) and ($<:$ reflexivity) the premises state $\Theta \vdash \Phi$, which implies consistency of Θ due to Lemma 4.18.

Induction cases: Suppose $\Theta \vdash \Phi <: \Psi$ has been derived using one of the rules (! left), (! right), (function subtype), (product subtype) or (sum subtype). In all these rules Θ appears as type context in all premises. Thus, by induction, we know Θ is consistent.

Finally, assume $\Theta \vdash \Phi <: \Psi$ has been derived using rule (polymorphic subtype). Then Θ is of the form Ξ, Ξ'. But since Ξ, Ξ' occurs as type context in the first premise, we may assume by induction that Θ is consistent.

□

Next, we show no occurrence of a type variable (be it free or bound) in the type term of a proved type stays without a bound on its type. Together with the definition of consistency and Lemma 4.18 this also means all type bounds on free variables in a proved type are also derived as part of a proved type. Despite of the meaning that it has on its own, it also opens the way to show how well-scopedness of type contexts relates to consistency.

Lemma 4.20. *If $\Theta \vdash \Phi$ is a proved type, then all free type variables in Φ appear in the domain of Θ, i.e. $ftyv(\Phi) \subseteq |\Theta|$.*

Proof. By induction on the derivation of $\Theta \vdash \Phi$:

Base cases: Suppose $\Theta \vdash \Phi$ has been derived using one of the rules (*Top* type), (*Unit* type) or (*Qbit* type). Then Φ has the form $!^n\,Top$, $!^n\,Unit$ or $Qbit$, i.e. Φ does not contain any free type variables at all.

Assume $\Theta \vdash \Phi$ has been derived using rule (linear type variable) or (nonlinear type variable), i.e. it is of the form $\Xi, X{<:}!^n\Phi_X, \Xi' \vdash X$ or $\Xi, X{<:}!^n\Phi_X, \Xi' \vdash !^n X$. Then we obviously have $ftyv(\Phi) = \{X\} \subseteq |\Xi, X{<:}!^n\Phi_X, \Xi'| = |\Xi, X{<:}!^n\Phi_X, \Xi'|$.

Induction cases: Suppose $\Theta \vdash \Phi$ is of the form $\Theta \vdash !^n(\Phi' \multimap \Psi')$ and has been derived using rule (function type). By induction we then know $ftyv(\Phi') \subseteq |\Theta|$ and also $ftyv(\Psi') \subseteq |\Theta|$. The definition of function *ftyv* tells us $ftyv(!^n(\Phi' \multimap \Psi')) = ftyv(\Phi') \cup ftyv(\Psi')$ which is hence a subset of $|\Theta|$. Similar arguments cover the cases of derivation rules (sum type) and (product type).

Assume $\Theta \vdash \Phi$ is of the form $\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi')$ and has been derived using rule (linear-polymorphic type). By induction we know $ftyv(\Psi') \subseteq |\Xi, X <: \Phi_X, \Xi'|$. From the definition of a type context and concatenation of type contexts we conclude $|\Xi, \Xi'| = |\Xi, X <: \Phi_X, \Xi'| \setminus \{X\}$. Moreover, we know (also by induction) that $ftyv(\Phi_X) \subseteq |\Xi|$, since we obtain proved type $\Xi \vdash \Phi_X$ by consistency of $\Xi, X{<:}\Phi_X, \Xi'$ which in turn is a consequence of Lemma 4.18 applied to the second premise of rule (linear-polymorphic type). The definition of function *ftyv* furthermore tells us $ftyv((\forall X{<:}\Phi_X.\Psi')) = ftyv(\Phi_X) \cup (ftyv(\Psi') \setminus \{X\})$. Hence, it holds $ftyv((\forall X{<:}\Phi_X.\Psi')) \subseteq |\Xi, \Xi'|$. The same line of argument can be applied in case of rule (nonlinear-polymorphic type).

□

Having established Lemma 4.20 it immediately follows that well-scopedness of type contexts is a weak form of consistency, or in other words:

Corollary 4.21. *A consistent type context is well-scoped.*

Proof. This is an immediate consequence of Lemma 4.20 and the definition of consistency for type contexts. □

Later on, we will learn a bit more about the connection between consistency and well-scopedness of subsequences of consistent type contexts, namely in subsection 4.2.2 and in particular in Lemma 4.27 and Corollary 4.28.

We can easily extend Lemma 4.20 to the setting of proved subtypes, i.e. from a proved subtype $\Theta \vdash \Phi <: \Psi$ we may conclude that all free type variables in Φ and Ψ are assigned a type bound in Θ, as the following lemma shows.

Lemma 4.22. *If $\Theta \vdash \Phi <: \Psi$ is a proved subtype, then $ftyv(\Phi) \cup ftyv(\Psi) \subseteq |\Theta|$.*

Proof. We perform an induction on the derivation of $\Theta \vdash \Phi <: \Psi$. For the base cases we use Lemma 4.20. We here only take a look at the "most complicated" induction case:
Let $\Theta \vdash \Phi <: \Psi$ be derived by

$$\frac{\Xi, \Xi' \vdash \Phi_X <: \Phi'_X \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi <: \Psi'}{\Xi, \Xi' \vdash (\forall X{<:}\Phi'_X.\Psi) <: (\forall X{<:}\Phi_X.\Psi')} \text{ (polymorphic subtype)} .$$

Induction yields $ftyv(\Phi_X) \cup ftyv(\Phi'_X) \subseteq |\Xi, \Xi'|$ and $ftyv(\Psi) \cup ftyv(\Psi') \subseteq |\Xi, X{<:}\Phi_X, \Xi'|$. Since

$$ftyv((\forall X{<:}\Phi'_X.\Psi)) \cup ftyv((\forall X{<:}\Phi_X.\Psi')) = \underbrace{ftyv(\Phi'_X) \cup ftyv(\Phi_X)}_{\subseteq\, |\Xi,\Xi'|} \cup \big(\underbrace{\big(ftyv(\Psi) \cup ftyv(\Psi')\big)}_{\subseteq\, |\Xi,\Xi'|\cup\{X\}} \setminus \{X\}\big) ,$$

we clearly have

$$ftyv((\forall X{<:}\Phi'_X.\Psi)) \cup ftyv((\forall X{<:}\Phi_X.\Psi')) \subseteq |\Xi, \Xi'| .$$

□

At this point we have finished a first exposition of the very basic characteristics that follow from the type and subtype derivation rules our system is based on. The results we have obtained so far will turn out to be very useful when we investigate more involved aspects. But before we jump into such ventures, we first need to develop some more sophisticated tools which open the way to a better understanding of the more advanced characteristics of our system.

4.2.2 Derived rules for proving types

In this subsection we derive some more rules for derivation of proved types and proved subtypes. These help us to perform derivations more concisely and thus also make some of the upcoming arguments much easier. Perhaps more important, however, they reveal some key characteristics of the type system.

In the beginning we will see that the order of variable-type pairs in type contexts does only matter to a certain extend, and that type contexts may be extended by certain additional pairs and still can be used to derive proved (sub)types that have already been derived without the additional information. These two properties are quite natural (although they do not appear automatically in all type systems), but since we have to keep a sharp eye on consistency of type contexts, we only get restricted versions of *permutation* and *weakening* of type contexts. On the other hand, type contexts may assign type bounds to type variables that are actually not needed for the derivation of a certain proved type. For such cases, we will see that we can remove some unnecessary variable-type pairs and still have enough information to derive proved (sub)types we have already proven with the "heavier" context. This can be regraded as *strengthening* of a type context (in contrast to weakening), since it narrows the assumptions we make on type variables. Moreover, we will learn which information about a type bound assigned to a type variable is really essential, and which is rather unimportant. This means we will see under which circumstances we can replace type bounds in a type context by other ones. The last aspect we will look at is the derivability of leading exponentials in proved types. To be more precise, we will see that we can add more leading exponentials if there is at least one in the beginning and that we can always remove them.

We start with type weakening, which as usual means adding more variable-type pairs to the type context of a proved type or subtype. But first, we need another basic notion:

Definition 4.23 (subsequence of a type context, $\sqsubseteq$ relation).
Let Θ and $\widehat{\Theta}$ be type contexts. We write $\Theta \sqsubseteq \widehat{\Theta}$, if $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$ is a subsequence of $\widehat{\Theta}$, i.e. $\widehat{\Theta} = \Xi_0, X_1{<:}\Phi_1, \Xi_1, \ldots, X_n{<:}\Phi_n, \Xi_n$, where $\Xi_0, \ldots, \Xi_n$ are arbitrary (possibly empty) type contexts.

Now we can quite comfortably formulate the weakening of type contexts in proved types and subtypes.

Proposition 4.24. *Let Θ and $\widehat{\Theta}$ be consistent type contexts with $\Theta \sqsubseteq \widehat{\Theta}$. The following rules can be derived from the ones in Definition 4.15:*

$$\frac{\vdash \widehat{\Theta} \qquad \Theta \vdash \Phi \qquad \Theta \sqsubseteq \widehat{\Theta}}{\widehat{\Theta} \vdash \Phi} \textit{ (type weakening) ,}$$

$$\frac{\vdash \widehat{\Theta} \qquad \Theta \vdash \Phi <: \Psi \qquad \Theta \sqsubseteq \widehat{\Theta}}{\widehat{\Theta} \vdash \Phi <: \Psi} \textit{ (subtype weakening) .}$$

Proof. We show the proposition by induction on the derivation of proved types and proved subtypes. Let Θ and $\widehat{\Theta}$ be two consistent type contexts such that $\Theta \sqsubseteq \widehat{\Theta}$. Additionally, let Θ' and $\widehat{\Theta}'$ be type contexts (not necessarily consistent) such that Θ, Θ' and $\widehat{\Theta}, \widehat{\Theta}'$ are consistent, and it holds $\Theta, \Theta' \sqsubseteq \widehat{\Theta}, \widehat{\Theta}'$. At first, we deal with the (type weakening) rule:

Base cases: Consider a proved type $\Theta \vdash \mathit{Qbit}$ derived using rule (*Qbit* type). Clearly, we can also derive proved type $\widehat{\Theta} \vdash \mathit{Qbit}$ using the same rule. This argument also holds for rules (*Top* type), (*Unit* type), (linear type variable) and (nonlinear type variable).

Induction cases: Suppose $\Theta \vdash !^n(\Phi' \multimap \Psi')$ has been derived using rule (function type). By induction, $\widehat{\Theta} \vdash \Phi'$ and $\widehat{\Theta} \vdash \Psi'$ are proved types. But then, we can also use (function type) to derive $\widehat{\Theta} \vdash !^n(\Phi' \multimap \Psi')$. A similar argument holds for rules (product type) and (sum type).

Assume proved type $\Theta, \Theta' \vdash (\forall X{<:}\Phi_X.\Psi')$ has been derived with rule (linear-polymorphic type). (In what follows, we additionally require $X \notin |\widehat{\Theta}, \widehat{\Theta}'|$, which is no real restriction since we identify α-equivalent type terms.) From the assumption we made in the beginning, we know $\widehat{\Theta}, \widehat{\Theta}'$ is consistent and from the third premise of the applied rule we know $X \notin \mathit{nftyv}(\Psi')$. Furthermore, we may conclude by induction that $\widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}' \vdash \Psi'$ is derivable, since clearly $\Theta, X{<:}\Phi_X, \Theta' \sqsubseteq \widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}'$, where both $\Theta, X{<:}\Phi_X, \Theta'$ and $\widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}'$ are consistent (the former one due to Lemma 4.18 and the latter one by iterated application of the induction hypothesis[30]). Now we have all necessary pieces together to apply rule (linear-polymorphic type) to obtain proved type $\widehat{\Theta}, \widehat{\Theta}' \vdash (\forall X{<:}\Phi_X.\Psi')$ as result.

We can follow a similar, slightly more elaborate argument to handle rule (nonlinear-polymorphic type).

$\Diamond$

Next, we come to rule (subtype weakening):

Base cases: Suppose $\Theta \vdash \Phi <: \Phi$ has been derived using rule ($<:$ reflexivity). Thus, premise $\Theta \vdash \Phi$ holds and using rule (type weakening), we may derive $\widehat{\Theta} \vdash \Phi$, which then enables us to derive $\widehat{\Theta} \vdash \Phi <: \Phi$ using ($<:$ reflexivity). We may argue similarly for rule (*Top* supertype).

Induction cases: Assume $\Theta \vdash \Phi <: \Psi$ has been derived using one of the rules (! left), (! right), (function subtype), (product subtype), (sum subtype). All necessary premises are of the form $\Theta \vdash \Phi' <: \Psi'$, from which we conclude by induction that $\widehat{\Theta} \vdash \Phi' <: \Psi'$ can be derived as proved subtype. But then, we can apply the respective rule to these new premises to obtain $\widehat{\Theta} \vdash \Phi <: \Psi$.

Finally, we argue along the same lines as for rules (linear-polymorphic type) and (nonlinear-polymorphic type) in the induction step for proving (type weakening). (And thus we assume $X \notin |\widehat{\Theta}, \widehat{\Theta}'|$ here, as well.) Suppose, $\Theta, \Theta' \vdash (\forall X <: \Phi'_X.\Psi') <: (\forall X <: \Phi_X.\Psi'')$ has been derived using rule (polymorphic subtype). From the second premise of this rule and the fact that $\Theta, X{<:}\Phi_X, \Theta' \sqsubseteq \widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}'$, where $\Theta, X{<:}\Phi_X, \Theta'$ and $\widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}'$ are consistent (the former one due to Lemma 4.19 and the latter one due to iterated application of rule (type weakening) as in the case of rule (linear-polymorphic type)), we conclude by induction that $\widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}' \vdash \Psi' <: \Psi''$ is a derivable proved subtype. Also by induction, we know $\widehat{\Theta}, \widehat{\Theta}' \vdash \Phi_X <: \Phi'_X$ is a proved subtype. This brings us to the point where we can derive $\widehat{\Theta}, \widehat{\Theta}' \vdash (\forall X{<:}\Phi'_X.\Psi') <: (\forall X{<:}\Phi_X.\Psi'')$ using rule (polymorphic subtype). $\square$

[30] This iteration goes step-by-step over each element of type context $\widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}'$, starting with $\vdash \widehat{\Theta} \ \wedge \ \Theta \vdash \Phi_X \overset{\text{induction}}{\Longrightarrow} \widehat{\Theta} \vdash \Phi_X$. To continue, we use consistency of $\widehat{\Theta}, \widehat{\Theta}'$ and inductively apply (type weakening) for each type bound in $\widehat{\Theta}' = Y_1{<:}\widehat{\Phi}_1, \ldots, Y_l{<:}\widehat{\Phi}_l$ one after another, i.e. $\vdash \widehat{\Theta}, X{<:}\Phi_X, Y_1{<:}\widehat{\Phi}_1, \ldots, Y_i{<:}\widehat{\Phi}_i \ \wedge \ \widehat{\Theta}, Y_1{<:}\widehat{\Phi}_1, \ldots, Y_i{<:}\widehat{\Phi}_i \vdash \widehat{\Phi}_{i+1} \ \Rightarrow \ \widehat{\Theta}, X{<:}\Phi_X, Y_1{<:}\widehat{\Phi}_1, \ldots, Y_i{<:}\widehat{\Phi}_i \vdash \widehat{\Phi}_{i+1}$. Proceeding in this way, we eventually ensure consistency of $\widehat{\Theta}, X{<:}\Phi_X, \widehat{\Theta}'$.

Next, we take a look at the order in which variable-type pairs appear in a type context. We can indeed change this order in the type context of a proved type or proved subtype as long as consistency is preserved for the new order.

Proposition 4.25. *Let $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$ be a consistent type context and let $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a bijective mapping (i.e. π is a permutation of the first n positive natural numbers), such that $\Theta^\pi := X_{\pi(1)}{<:}\Phi_{\pi(1)}, \ldots, X_{\pi(n)}{<:}\Phi_{\pi(n)}$ is a consistent type context containing the same variable-type pairs as Θ does, permuted according to π. Then the following rules can be derived:*

$$\frac{\vdash \Theta^\pi \qquad \Theta \vdash \Phi}{\Theta^\pi \vdash \Phi}\ \textit{(type permutation)}\ , \qquad \frac{\vdash \Theta^\pi \qquad \Theta \vdash \Phi <: \Psi}{\Theta^\pi \vdash \Phi <: \Psi}\ \textit{(subtype permutation)}\ .$$

Proof. We prove the first rule by induction on the derivation of proved type $\Theta \vdash \Phi$. Using this result, we prove the other rule by induction on the derivation of proved subtype $\Theta \vdash \Phi <: \Psi$.

Let Θ be a consistent type context and Θ^π be a consistent permuted version of Θ as described above. (The same shall hold for consistent type contexts $\Theta, X{<:}\Phi_X, \Theta'$ and $(\Theta, X{<:}\Phi_X, \Theta')^\pi$, and Θ, Θ' and $(\Theta, \Theta')^\pi$ in the respective cases.)

We start with rule (type permutation):

Base cases: Suppose $\Theta \vdash \mathit{Qbit}$ has been derived using rule (*Qbit* type). But then $\Theta^\pi \vdash \mathit{Qbit}$ can be derived using the same rule, since Θ^π is also consistent. The same obviously holds for rules (*Top* type) and (*Unit* type). Rules (linear type variable) and (nonlinear type variable) can be treated in the same way, although the type contexts in question are of the form $\Theta, X{<:}\Phi_X, \Theta'$ and $(\Theta, X{<:}\Phi_X, \Theta')^\pi$.

Induction cases: At first, suppose $\Theta \vdash !^n(\Phi' \multimap \Psi')$ has been derived using rule (function type) in the last step. By induction we may assume $\Theta^\pi \vdash \Phi'$ and $\Theta^\pi \vdash \Psi'$ are proved types. Then rule (function type) is also applicable to these premises leading to proved type $\Theta^\pi \vdash !^n(\Phi' \multimap \Psi')$. Again, similar arguments also hold for rules (product type) and (sum type).

As a second step, assume $\Theta, \Theta' \vdash (\forall X <: \Phi_X.\Psi')$ is a proved type derived using rule (linear-polymorphic type). On the one hand, we know from the premises of this rule that $X \notin \mathit{nftyv}(\Psi')$. On the other hand, we may conclude by induction that $(\Theta, \Theta')^\pi, X{<:}\Phi_X \vdash \Psi'$ is a proved type, since $(\Theta, \Theta')^\pi, X{<:}\Phi_X$ is a permuted and consistent version of $\Theta, X{<:}\Phi_X, \Theta'$. This follows from consistency of $\Theta, X{<:}\Phi_X, \Theta'$ (due the second premise of rule (linear-polymorphic type) and Lemma 4.18) which entails $\Theta, X{<:}\Phi_X$ is consistent, i.e. $\Theta \vdash \Phi_X$ can be derived. This leads to a derivation of $\Theta, \Theta' \vdash \Phi_X$ by (type weakening), and hence to $(\Theta, \Theta')^\pi \vdash \Phi_X$ by induction, which yields consistency of $(\Theta, \Theta')^\pi, X{<:}\Phi_X$. Having all this, together with the assumption of $(\Theta, \Theta')^\pi$ being consistent, we apply rule (linear-polymorphic type) to obtain proved type $(\Theta, \Theta')^\pi \vdash (\forall X{<:}\Phi_X.\Psi')$.

Finally, suppose proved type $\Theta, \Theta' \vdash !^n(\forall X{<:}\Phi_X.\Psi')$ has been derived using rule (nonlinear-polymorphic type). We employ a similar argument as before, and thus use the respective premises of this rule to derive proved type $(\Theta, \Theta')^\pi, X{<:}\Phi_X \vdash \Psi'$ by induction, Lemma 4.18, Lemma 4.19 and rule (type weakening). In addition, $(\Theta, \Theta')^\pi \vdash \Phi_X <: !\mathit{Top}$ can be derived by

$$\dfrac{\dfrac{(\Theta, \Theta')^\pi \vdash !\Phi'_X}{(\Theta, \Theta')^\pi \vdash !\Phi'_X <: \mathit{Top}}\ (\mathit{Top}\text{ supertype})}{(\Theta, \Theta')^\pi \vdash !\Phi'_X <: !\mathit{Top}}\ (!\text{ right})$$

where Φ_X is of the form $!\Phi'_X$ for some $\Phi'_X \in \mathcal{T}_{type}$ (according to Corollary 4.17 applied to premise $\Theta, \Theta' \vdash \Phi_X <: !\mathit{Top}$), and where $(\Theta, \Theta')^\pi \vdash \Phi_X$ holds by induction. Using $(\Theta, \Theta')^\pi, X{<:}\Phi_X \vdash \Psi'$ and $(\Theta, \Theta')^\pi \vdash \Phi_X <: !\mathit{Top}$ as premises, we may eventually derive $(\Theta, \Theta')^\pi \vdash !^n(\forall X{<:}\Phi_X.\Psi')$ by rule (nonlinear-polymorphic type).

◊

Next, we treat rule (subtype permutation):

Base cases: We assume $\Theta \vdash \Phi <: \mathit{Top}$ has been derived using rule (*Top* supertype). From its premise $\Theta \vdash \Phi$ we can derive proved type $\Theta^\pi \vdash \Phi$ using the just proven rule (type permutation). By applying rule (*Top* supertype) to this, we then get proved subtype $\Theta^\pi \vdash \Phi <: \mathit{Top}$. The same argument holds for rule ($<:$ reflexivity).

Induction cases: Suppose proved subtype $\Theta \vdash (\Phi'' \multimap \Psi') <: (\Phi' \multimap \Psi'')$ has been derived by rule (function subtype). Induction applied to the two premises leads to proved subtypes $\Theta^\pi \vdash \Phi' <: \Phi''$ and $\Theta^\pi \vdash \Psi' <: \Psi''$. Now we can immediately apply rule (function subtype) again to get proved subtype $\Theta^\pi \vdash (\Phi'' \multimap \Psi') <: (\Phi' \multimap \Psi'')$. Similar arguments hold for rules (! left), (! right), (product subtype) and (sum subtype).

A slightly more interesting case is the one concerned with rule (polymorphic subtype). Assume proved subtype $\Theta, \Theta' \vdash (\forall X <: \Phi'_X.\Psi') <: (\forall X <: \Phi_X.\Psi'')$ has been derived using this rule. By induction we conclude $(\Theta, \Theta')^\pi \vdash \Phi_X <: \Phi'_X$ is a proved subtype. Lemma 4.19 applied to the first and second premises yields consistency of Θ, Θ' and $\Theta, X <: \Phi_X, \Theta'$, respectively. From this we conclude that $\Theta \vdash \Phi_X$ is a proved type, which we may weaken to $\Theta, \Theta' \vdash \Phi_X$. Using rule (type permutation) we obtain $(\Theta, \Theta')^\pi \vdash \Phi_X$ from this, which immediately translates to consistency of $(\Theta, \Theta')^\pi, X <: \Phi_X$, i.e. a permuted version of $\Theta, X <: \Phi_X, \Theta'$. Having established this, induction now leads us from the second premise of (polymorphic subtype) to proved subtype $(\Theta, \Theta')^\pi, X <: \Phi_X \vdash \Psi' <: \Psi''$. This finally enables the derivation of $(\Theta, \Theta')^\pi \vdash (\forall X <: \Phi'_X.\Psi') <: (\forall X <: \Phi_X.\Psi'')$ as a proved subtype using rule (polymorphic subtype).

□

By now, we have seen the possibilities of how to weaken and permute type contexts of a proved type. Henceforth, we will very often make use of rules (type permutation) and (type weakening), especially of the latter one.

As a counterpoint to this we now take a look at the strengthening or narrowing of a type context in a proved type. Therefore, we first introduce a notion and appropriate notations expressing the restriction of the domain of a type context to a certain set of type variables, i.e. to a subset of $\mathcal{V}_{type}$.

Definition 4.26 (restriction of type contexts).

Let $\Theta = X_1 <: \Phi_1, \ldots, X_n <: \Phi_n$ be an arbitrary type context of length $n \geq 0$. We write $\Theta(X_i)$ to address type bound Φ_i for $1 \leq i \leq n$. In case of $X \notin |\Theta|$, we write $\Theta(X) = \bot$.

Let $V \subseteq \mathcal{V}_{type}$ be a (possibly empty) set of type variables. We write $\Theta|_V$ for the *restriction of* Θ *to the type variables in* V, which we define as subsequence $\Theta|_V \sqsubseteq \Theta$ such that

$$\Theta|_V(X) := \begin{cases} \Theta(X) & \text{, if } X \in V, \\ \bot & \text{, if } X \notin V. \end{cases}$$

Now we have the right tools at hand to obtain the result which justifies strengthening of type contexts in proved types. From Lemma 4.20 we know that we have to keep all type variables in the domain of the type context which occur freely in the type term right of the turnstile symbol "$\vdash$". Furthermore, Lemma 4.18 tells us that the restricted type context needs to be consistent, which immediately implies well-scopedness due to Corollary 4.21. We use this knowledge to formulate the appropriate requirements in the following lemma, i.e. keep free type variables and stay well-scoped. We will see this is already enough.

Lemma 4.27. *Let $\Theta \vdash \Phi$ be a proved type and let $V \subseteq \mathcal{V}_{type}$ be a set of type variables which fulfills*

(1) $\mathit{ftyv}(\Phi) \subseteq V$,

(2) for all $X \in V$ it holds $\mathit{ftyv}\big(\Theta(X)\big) \subseteq V$.[31]

Then $\Theta|_V \vdash \Phi$ is also derivable as a proved type. (For convenience, we here define $\mathit{ftyv}(\bot) := \emptyset$.)

Proof. We show this by nested induction on the length of type context Θ (outer induction) and on the derivation of $\Theta \vdash \Phi$ (inner induction).

[31] This second criterion aims at nothing else than well-scopedness of $\Theta|_V$.

Outer base case: Let Θ be the empty type context, i.e. $\Theta = \emptyset$. Since $\emptyset|_V = \emptyset$ for any $V \subseteq \mathcal{V}_{type}$, the lemma trivially holds in this case.

Outer induction case: Let $\Theta = \Theta', Z{<:}\Phi_Z$ be a nonempty type context with subsequence $\Theta' := Y_1{<:}\Phi_{Y_1}, \ldots, Y_l{<:}\Phi_{Y_l}$ of length $l \geq 0$.

Inner base cases: Assume $\Theta \vdash !^n \mathit{Top}$ has been derived using rule (*Top* type). By Lemma 4.18 it then follows that Θ is consistent. That means we have proved types

$$\begin{aligned} \vdash \Phi_{Y_1} \ , \\ Y_1{<:}\Phi_{Y_1} \vdash \Phi_{Y_2} \ , \\ Y_1{<:}\Phi_{Y_1}, Y_2{<:}\Phi_{Y_2} \vdash \Phi_{Y_3} \ , \\ \vdots \\ Y_1{<:}\Phi_{Y_1}, \ldots, Y_{l-1}{<:}\Phi_{Y_{l-1}} \vdash \Phi_{Y_l} \ , \\ Y_1{<:}\Phi_{Y_1}, \ldots, Y_{l-1}{<:}\Phi_{Y_{l-1}}, Y_l{<:}\Phi_{Y_l} \vdash \Phi_Z \ . \end{aligned}$$

For each of the $Y_i \in V \cap \{Y_1, \ldots, Y_l\}$ we have

(1') $\mathit{ftyv}(\Phi_{Y_i}) \subseteq V$ due to (2), and

(2') $\mathit{ftyv}\big(\Theta|_{\{Y_1,\ldots,Y_{i-1}\}}(X)\big) \subseteq V$ for all $X \in V$, since we have $\Theta|_{\{Y_1,\ldots,Y_{i-1}\}} \sqsubseteq \Theta$ and due to (2).

Hence, we may apply outer induction to obtain proved type

$$(Y_1{<:}\Phi_{Y_1}, \ldots, Y_{i-1}{<:}\Phi_{Y_{i-1}})|_V \vdash \Phi_{Y_i}$$

for each $Y_i \in V \cap \{Y_1, \ldots, Y_l\}$.

In case of $Z \in V$, we analogously have

(1") $\mathit{ftyv}(\Phi_Z) \subseteq V$ and

(2") $\mathit{ftyv}\big(\Theta|_{\{Y_1,\ldots,Y_l\}}(X)\big) \subseteq V$.

Thus, outer induction yields $(Y_1{<:}\Phi_{Y_1}, \ldots, Y_l{<:}\Phi_{Y_l})|_V \vdash \Phi_Z$ in this case.

Hence, we know $\Theta|_V$ is consistent. But then we can immediately apply rule (*Top* type) to derive $\Theta|_V \vdash !^n \mathit{Top}$. We may argue analogously to treat the cases of derivation rules (*Unit* type) and (*Qbit* type).

Suppose $\Theta \vdash !^n X$ has been derived using rule (linear type variable) or (nonlinear type variable). Analogously to the case of rule (*Top* type), we can show consistency of $\Theta|_V$. Since we assume $\mathit{ftyv}(\Phi) = \mathit{ftyv}(!^n X) \subseteq V$ in (1), we thus have $\Theta|_V = \Xi, X{<:}\Phi_X, \Xi'$ for some type contexts Ξ and Ξ'. But then, we may apply the respective rule (linear type variable) or (nonlinear type variable) to derive proved type $\Theta|_V \vdash !^n X$.

Inner induction cases: Suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\Theta \vdash !^n \Phi' \qquad \Theta \vdash !^n \Psi'}{\Theta \vdash !^n(\Phi' \otimes \Psi')} \text{ (product type)} \ .$$

Inner induction (where the necessary requirements are fulfilled, since $\mathit{ftyv}(\Phi'), \mathit{ftyv}(\Psi') \subseteq \mathit{ftyv}((\Phi' \otimes \Psi')) \subseteq V$ holds by (1), and due to (2)) then tells us that both $\Theta|_V \vdash !^n \Phi'$ and $\Theta|_V \vdash !^n \Psi'$ are proved types. Hence, we may apply (product type) to derive proved type $\Theta|_V \vdash !^n(\Phi' \otimes \Psi')$. The same line of argument applies to the cases of rules (function type) and (sum type).

Assume $\Theta \vdash \Phi$ has been derived by

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi')} \text{ (linear-polymorphic type)} \ ,$$

where we have $\Xi, \Xi' = \Theta', Z <: \Phi_Z$. As before, we conclude consistency of $(\Xi, \Xi')|_V$ from premise $\vdash \Xi, \Xi'$ by outer induction (analogous to the case of rule (*Top* type)). Since type context $\Xi, X{<:}\Phi_X, \Xi'$ in the second premise is considered to be properly defined, we conclude $X \notin |\Xi, \Xi'|$. By definition of restrictions of type contexts, we thus know $(\Xi, \Xi')|_V = (\Xi, \Xi')|_{V \cup \{X\}}$.

Moreover, we have $\mathit{ftyv}(\Psi') \subseteq \mathit{ftyv}((\forall X{<:}\Phi_X.\Psi')) \cup \{X\} \subseteq V \cup \{X\}$, which holds by (1), on the one hand. On the other hand, $\mathit{ftyv}\big((\Xi, X{<:}\Phi_X, \Xi')(Y)\big) \subseteq V \cup \{X\}$ holds for all $Y \in V \cup \{X\}$ due to (2) and the fact that $\mathit{ftyv}(\Phi_X) \subseteq \mathit{ftyv}((\forall X{<:}\Phi_X.\Psi')) \subseteq V$ holds by (1). Hence, we may use inner induction to conclude from the second premise that $(\Xi, X{<:}\Phi_X, \Xi')|_{V\cup\{X\}} \vdash \Psi'$ is a proved type. Having all this, we can perform derivation

$$\frac{\vdash (\Xi,\Xi')|_{V\cup\{X\}} \qquad (\Xi, X{<:}\Phi_X, \Xi')|_{V\cup\{X\}} \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{(\Xi,\Xi')|_{V\cup\{X\}} \vdash (\forall X{<:}\Phi_X.\Psi')}\ \text{(linear-polymorphic type)}\ ,$$

by which we obtain the proved type $(\Xi,\Xi')|_V \vdash (\forall X{<:}\Phi_X.\Psi')$ (since $(\Xi,\Xi')|_V = (\Xi,\Xi')|_{V\cup\{X\}}$, as we have shown above).
Finally, suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\Xi,\Xi' \vdash \Phi_X <: !\mathit{Top} \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash !^n\Psi'}{\Xi,\Xi' \vdash !^n(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}$$

with $\Xi,\Xi' = \Theta', Z{<:}\Phi_Z$. Applying Lemma 4.18 to the second premise, we get consistency of $\Xi, X{<:}\Phi_X, \Xi'$, and thus know $\Xi \vdash \Phi_X$ is a proved type. This may be weakened to $\Xi,\Xi' \vdash \Phi_X$, since we get consistency of Ξ,Ξ' by Lemma 4.19 applied to the first premise. By inner induction (where we know $\mathit{ftyv}(\Phi_X) \subseteq \mathit{ftyv}((\forall X{<:}\Phi_X.\Psi')) \subseteq V$ by (1), and (2) ensures that the second requirement in fulfilled), we may then conclude $(\Xi,\Xi')|_{V\cup\{X\}} \vdash \Phi_X$ is a proved type, as well. (We have already argued above, that $(\Xi,\Xi')|_{V\cup\{X\}} = (\Xi,\Xi')|_V$ holds in this case.) Moreover, we know $\Phi_X = !\Phi'_X$ holds for some $\Phi'_X \in \mathcal{T}_{\mathit{type}}$ due to Corollary 4.17 applied to the first premise. Hence, we derive $(\Xi,\Xi')|_{V\cup\{X\}} \vdash \Phi_X <: !\mathit{Top}$ by

$$\frac{\dfrac{(\Xi,\Xi')|_{V\cup\{X\}} \vdash !\Phi'_X}{(\Xi,\Xi')|_{V\cup\{X\}} \vdash !\Phi'_X <: \mathit{Top}}\ (\mathit{Top}\text{ supertype})}{(\Xi,\Xi')|_{V\cup\{X\}} \vdash !\Phi'_X <: !\mathit{Top}}\ (!\text{ right})\ .$$

Analogously to the previous induction case of rule (linear-polymorphic type), we get the last missing piece from inner induction applied to the second premise, which yields proved type $(\Xi, X{<:}\Phi_X, \Xi')|_{V\cup\{X\}} \vdash !^n\Psi'$. All this together enables derivation

$$\frac{(\Xi,\Xi')|_{V\cup\{X\}} \vdash \Phi_X <: !\mathit{Top} \qquad (\Xi, X{<:}\Phi_X, \Xi')|_{V\cup\{X\}} \vdash !^n\Psi'}{(\Xi,\Xi')|_{V\cup\{X\}} \vdash !^n(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}$$

by which we in the end obtain proved type $(\Xi,\Xi')|_V \vdash !^n(\forall X{<:}\Phi_X.\Psi')$.

□

When we combine the just proven lemma with the statement of Lemma 4.18, we get an immediate consequence: a well-scoped subsequence of a consistent type context is consistent. We will use this later on, especially in subsection 4.2.4 for showing some results concerning minimal type contexts for proved types and proved subtypes. Hence, it becomes part of the following corollary which besides this consequence contains a formulation of the preceding lemma that is nearer to actual application in future arguments than the original lemma is.

Corollary 4.28.

(i) Let $\widehat{\Theta} \vdash \Phi$ be a proved type. If Θ is a well-scoped type context with $\Theta \sqsubseteq \widehat{\Theta}$ and $\mathit{ftyv}(\Phi) \subseteq |\Theta|$, then $\Theta \vdash \Phi$ is derivable as proved type.

(ii) Let $\widehat{\Theta}$ be a consistent type context. If Θ is a well-scoped type context with $\Theta \sqsubseteq \widehat{\Theta}$, then Θ is consistent.

Proof.

(i): We apply Lemma 4.27 to show this:
Clearly, type context Θ with $\Theta \sqsubseteq \widehat{\Theta}$ can be seen as $\widehat{\Theta}|_{|\Theta|}$, where $\mathit{ftyv}(\Phi) \subseteq |\Theta|$ is assumed to hold. Recalling the definition of well-scopedness of type contexts (Definition 4.12), we immediately see that from well-scopedness of Θ it follows for all $X \in |\Theta|$ that $\mathit{ftyv}\big(\Theta(X)\big) \subseteq |\Theta|$, and thus also $\mathit{ftyv}\big(\widehat{\Theta}(X)\big) \subseteq |\Theta|$ by definition of the $\sqsubseteq$ relation (Definition 4.23). Hence, all requirements of Lemma 4.27 are fulfilled and thus, $\Theta \vdash \Phi$ is derivable.

(ii): The case of $\widehat{\Theta}$ being the empty type context is trivial, since then we have $\widehat{\Theta} = \Theta$.
Let $\widehat{\Theta} = X_1{<:}\widehat{\Phi}_1, \ldots, X_n{<:}\widehat{\Phi}_k$ be of length $k \geq 1$ and let $Y{<:}\Phi_Y$ be the rightmost variable-type pair in Θ, i.e. $\Theta = \Theta', Y{<:}\Phi_Y$ for some type context Θ'. By definition of well-scopedness, Θ' is also well-scoped and hence we have $\mathit{ftyv}(\Phi_Y) \subseteq |\Theta'|$. Since Θ is a subsequence of $\widehat{\Theta}$, there must be an index i with $1 \leq i \leq k$, such that $Y = X_i$ and $\Phi_Y = \widehat{\Phi}_i$. We define $\widehat{\Theta}'$ to contain the first $i-1$ variable-type pairs of $\widehat{\Theta}$, i.e. $\widehat{\Theta}' := X_1{<:}\widehat{\Phi}_1, \ldots, X_{i-1}{<:}\widehat{\Phi}_{i-1}$. By definition of consistency, we know $\widehat{\Theta}' \vdash \widehat{\Phi}_i = \widehat{\Theta}' \vdash \Phi_Y$ is a proved type. Then, by (i), we can derive $\Theta' \vdash \Phi_Y$. Hence, $\Theta = \Theta', Y{<:}\Phi_Y$ is consistent.

□

We can easily extend Corollary 4.28(i) to the setting of proved subtypes:

Corollary 4.29. *Let $\widehat{\Theta} \vdash \Phi <: \Psi$ be a proved subtype. If Θ is a well-scoped type context with $\Theta \sqsubseteq \widehat{\Theta}$ and $\mathit{ftyv}(\Phi) \cup \mathit{ftyv}(\Psi) \subseteq |\Theta|$, then $\Theta \vdash \Phi <: \Psi$ is a derivable proved subtype.*

Proof. By induction on the derivation of $\widehat{\Theta} \vdash \Phi <: \Psi$.

Base cases: Suppose $\widehat{\Theta} \vdash \Phi <: \Psi$ has been derived by (*Top* supertype) or (<: reflexivity). Thus, we have $\widehat{\Theta} \vdash \Phi$ as premise. Then, by Corollary 4.28(i), we may derive $\Theta \vdash \Phi$, which leads to $\Theta \vdash \Phi <: \mathit{Top}$ and $\Theta \vdash \Phi <: \Phi$ when we use rules (*Top* supertype) and (<: reflexivity), respectively.

Induction cases: Suppose $\widehat{\Theta} \vdash \Phi <: \Psi$ has been derived by one of the rules (! left), (! right), (function type), (product type) or (sum type). Then all of the premises have the form $\widehat{\Theta} \vdash \Phi' <: \Psi'$. By induction we can derive $\Theta \vdash \Phi' <: \Psi'$ for all of these proved subtypes. (Induction is possible in these cases, since we have $\mathit{ftyv}(\Phi') \cup \mathit{ftyv}(\Psi') \subseteq \mathit{ftyv}(\Phi) \cup \mathit{ftyv}(\Psi)$ in all cases, which we see by inspection of the definition of function *ftyv*.) Then we can apply the respective derivation rule again to obtain proved subtype $\Theta \vdash \Phi <: \Psi$.

Now assume the somewhat more interesting case in which $\widehat{\Theta} \vdash \Phi <: \Psi$ has been derived by

$$\frac{\widehat{\Theta} \vdash \Psi_1 <: \Phi_1 \qquad \widehat{\Xi}, X{<:}\Psi_1, \widehat{\Xi}' \vdash \Phi_2 <: \Psi_2}{\widehat{\Theta} \vdash (\forall X{<:}\Phi_1.\Phi_2) <: (\forall X{<:}\Psi_1.\Psi_2)}\ \text{(polymorphic subtype)}$$

with $\widehat{\Theta} = \widehat{\Xi}, \widehat{\Xi}'$. Lemma 4.19 yields consistency of $\widehat{\Theta} = \widehat{\Xi}, \widehat{\Xi}'$ (from the first premise) and $\widehat{\Xi}, X{<:}\Psi_1, \widehat{\Xi}'$ (from the second premise). The latter consistency result gives us proved type $\widehat{\Xi} \vdash \Psi_1$. By derivation rule (type weakening) and the fact $\widehat{\Xi} \sqsubseteq \widehat{\Theta}$ we then get proved type $\widehat{\Theta} \vdash \Psi_1$ and thus consistency of $\widehat{\Theta}, X{<:}\Psi_1$. With the help of rule (type permutation) we further obtain proved subtype $\widehat{\Theta}, X{<:}\Psi_1 \vdash \Phi_2 <: \Psi_2$ from $\widehat{\Xi}, X{<:}\Psi_1, \widehat{\Xi}' \vdash \Phi_2 <: \Psi_2$ where we use $\widehat{\Theta} = \widehat{\Xi}, \widehat{\Xi}'$ again.

By consistency of type context $\widehat{\Theta}, X{<:}\Psi_1$ (which is also well-scoped due to Corollary 4.21) we get proved type $\widehat{\Theta} \vdash \Psi_1$, from which we get $\Theta \vdash \Psi_1$ by Corollary 4.28(i), and which then again entails consistency of $\Theta, X{<:}\Psi_1$. And since

$$\begin{aligned}\mathit{ftyv}(\Psi_1) \cup \mathit{ftyv}(\Phi_1) \cup \big(\big(\mathit{ftyv}(\Phi_2) \cup \mathit{ftyv}(\Psi_2)\big) \setminus \{X\}\big) \\ = \mathit{ftyv}((\forall X{<:}\Phi_1.\Phi_2)) \cup \mathit{ftyv}((\forall X{<:}\Psi_1.\Psi_2)) \\ \subseteq |\Theta|\end{aligned}$$

holds by assumption, we also know $\mathit{ftyv}(\Phi_2) \cup \mathit{ftyv}(\Psi_2) \subseteq |\Theta, X <: \Psi_1|$. Hence, we may apply induction to $\widehat{\Theta}, X{<:}\Psi_1 \vdash \Phi_2 <: \Psi_2$, which leads to proved subtype $\Theta, X{<:}\Psi_1 \vdash \Phi_2 <: \Psi_2$. Induction applied to the first premise moreover yields the proved subtype $\Theta \vdash \Psi_1 <: \Phi_1$. Consequently, we may perform derivation

$$\frac{\Theta \vdash \Psi_1 <: \Phi_1 \qquad \Theta, X{<:}\Psi_1 \vdash \Phi_2 <: \Psi_2}{\Theta \vdash (\forall X{<:}\Phi_1.\Phi_2) <: (\forall X{<:}\Psi_1.\Psi_2)}\ \text{(polymorphic subtype)}\ .$$

□

The next issue we focus on is the following question: what information inside the type context of a proved type is essential and what information is dispensable? With the next derived rule we come to a remarkable and quite useful answer to this question. Let $\Theta \vdash \Phi$ be a proved type with type context $\Theta = Y_1{<:}\Phi_1, \ldots, Y_l{<:}\Phi_l$. It turns out that the essential information in Θ is whether type bounds Φ_i with $1 \leq i \leq n$ are linear or duplicable type terms. This means we can replace any Φ_i with Φ' as long as Φ' has at least one leading exponential if Φ_i has at least one and as long as Φ' is derivable as part of a proved type $Y_1{<:}\Phi_1, \ldots, Y_{i-1}{<:}\Phi_{i-1} \vdash \Phi'$ as the following lemma shows.[32]

Lemma 4.30. *We can derive rule*

$$\frac{\Theta \vdash !^m\Upsilon \qquad \Theta, Y{<:}!^n\Phi_Y, \Theta' \vdash \Psi \qquad \begin{array}{c}\Phi_Y, \Upsilon \text{ linear} \\ n > 0 \Rightarrow m > 0\end{array}}{\Theta, Y{<:}!^m\Upsilon, \Theta' \vdash \Psi}\ \textit{(type bound replacement)}\ , \qquad (\star)$$

where premise "Φ_Y, Υ linear" requires type terms Φ_Y and Υ to be linear, in other words, not to contain any leading exponentials; and premise "$n > 0 \Rightarrow m > 0$" means $n > 0$ implies $m > 0$.

Proof. By nested induction on the length of type context Θ' (outer induction) and the derivation of $\Theta, Y{<:}!^n\Phi_Y, \Theta' \vdash \Psi$ (inner induction):

Outer base case: Assume Θ' is the empty type context.

Inner base cases: Suppose $\Theta, Y{<:}!^n\Phi_Y \vdash !^k \mathit{Top}$, $k \geq 0$, has been derived using rule (*Top* type). Due to premise $\Theta \vdash !^m\Upsilon$ in $(\star)$, we know $\Theta, Y{<:}!^m\Upsilon$ is consistent and thus we can use rule (*Top* type) as before to derive $\Theta, Y{<:}!^m\Upsilon \vdash !^k \mathit{Top}$. Similar arguments cover cases (*Unit* type) and (*Qbit* type).

Now suppose $\Theta, Y{<:}!^n\Phi_Y \vdash \Psi$ has been derived by

$$\frac{\vdash \Theta, Y{<:}!^n\Phi_Y}{\Theta, Y{<:}!^n\Phi_Y \vdash X}\ \text{(linear type variable)}\ .$$

We distinguish two cases:

- $X \neq Y$: Then, Θ has the form $\Theta = \Xi, X{<:}\Phi_X, \Xi'$. Of course, we may then use consistent type context $\Xi, X{<:}\Phi_X, \Xi', Y{<:}!^m\Upsilon = \Theta, Y{<:}!^m\Upsilon$ (which is consistent due to the first premise in $(\star)$) to derive $\Theta, Y{<:}!^m\Upsilon \vdash X$ using rule (linear type variable).
- $X = Y$: Due to the first premise in $(\star)$, we have consistency of $\Theta, Y{<:}!^m\Upsilon$. But then we can perform derivation

$$\frac{\vdash \Theta, Y{<:}!^m\Upsilon}{\Theta, Y{<:}!^m\Upsilon \vdash Y}\ \text{(linear type variable)}\ .$$

Next, assume $\Theta, Y{<:}!^n\Phi_Y \vdash \Psi$ has been derived by

$$\frac{\vdash \Theta, Y{<:}!^n\Phi_Y}{\Theta, Y{<:}!^n\Phi_Y \vdash !^k X}\ \text{(nonlinear type variable)}\ .$$

By definition of this derivation rule, we know $n \geq 1$. Again, we need to distinguish two cases:

- $X \neq Y$: Then Θ has the form $\Theta = \Xi, X{<:}!\Phi_X, \Xi'$. Consequently, we may use consistent type context $\Xi, X{<:}!\Phi_X, \Xi', Y{<:}!^m\Upsilon$ (consistent due to the first premise in $(\star)$) to derive $\Theta, Y{<:}!^m\Upsilon \vdash !^k X$ using rule (nonlinear type variable).
- $X = Y$: Since we have $n \geq 1$, premise $n > 0 \Rightarrow m > 0$ in $(\star)$ entails $m \geq 1$. Hence, we can perform derivation

$$\frac{\vdash \Theta, Y{<:}!^m\Upsilon}{\Theta, Y{<:}!^m\Upsilon \vdash !^k Y}\ \text{(nonlinear type variable)}\ .$$

[32] This last condition is in fact formulated a bit too strong. If we employ more of the results we have worked out so far in this section and if we iteratively apply the replacement we are currently talking about, then we can also use the replacement inside type context $Y_1{<:}\Phi_1, \ldots, Y_{i-1}{<:}\Phi_{i-1}$ that has been used to derive Φ_i, and then use this modified type context to derive a proved type with Φ' right of $\vdash$.

Inner induction cases: Assume $\Theta, Y{<:}!^n\Phi_Y \vdash !^k(\Phi' \otimes \Psi')$ has been derived using rule (product type). In this case we may conclude by inner induction that $\Theta, Y{<:}!^m\Upsilon \vdash !^k\Phi'$ and $\Theta, Y{<:}!^m\Upsilon \vdash !^k\Psi'$ are derivable. But then we may also derive $\Theta, Y{<:}!^m\Upsilon \vdash !^k(\Phi' \otimes \Psi')$ using (product type). Similar arguments cover the cases of rules (sum type) and (function type).

Assume $\Theta, Y{<:}!^n\Phi_Y \vdash \Psi$ has been derived by

$$\frac{\vdash \Xi, \Xi', Y{<:}!^n\Phi_Y \qquad \Xi, X{<:}\Phi_X, \Xi', Y{<:}!^n\Phi_Y \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi', Y{<:}!^n\Phi_Y \vdash (\forall X{<:}\Phi_X.\Psi')}\ \text{(linear-polymorphic type)}\ ,$$

where we write Θ as Ξ, Ξ'. It is clear that $X \neq Y$, since otherwise the type context of the second premise would not be a properly defined type context. Using induction (applied to the second premise) and the first premise of ($\star$), we can perform derivation

$$\frac{\vdash \Xi, \Xi', Y{<:}!^m\Upsilon \qquad \Xi, X{<:}\Phi_X, \Xi', Y{<:}!^m\Upsilon \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi', Y{<:}!^m\Upsilon \vdash (\forall X{<:}\Phi_X.\Psi')}\ \text{(linear-polymorphic type)}$$

instead of the former one to show $\Theta, Y{<:}!^m\Upsilon \vdash (\forall X{<:}\Phi_X.\Psi')$ is a proved type.

The situation for rule (nonlinear-polymorphic type) is quite similar:

$$\frac{\Xi, \Xi', Y{<:}!^n\Phi_Y \vdash \Phi_X <: !\mathit{Top} \qquad \Xi, X{<:}\Phi_X, \Xi', Y{<:}!^n\Phi_Y \vdash !^k\Psi'}{\Xi, \Xi', Y{<:}!^n\Phi_Y \vdash !^k(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}\ .$$

However, we there have to show that $\Xi, \Xi', Y{<:}!^m\Upsilon \vdash \Phi_X <: !\mathit{Top}$ is derivable. But this is not hard to do starting from premise $\Xi, X{<:}\Phi_X, \Xi', Y{<:}!^n\Phi_Y \vdash !^k\Psi'$ to gain $\Xi \vdash \Phi_X$ by Lemma 4.18. Then we derive $\Xi \vdash \Phi_X <: !\mathit{Top}$ using rules (*Top* supertype) and (! right), where $\Phi_X = !\Phi'_X$ (for a certain type term Φ'_X) follows from Corollary 4.17 applied to the first premise of (nonlinear-polymorphic type). And since consistency of $\Xi, \Xi', Y{<:}!^m\Upsilon$ is ensured by the first premise of ($\star$), we may use (subtype weakening) to finally obtain $\Xi, \Xi', Y{<:}!^m\Upsilon \vdash !\Phi'_X <: !\mathit{Top}$. Then we continue analogous to the previous case to finally come up with derivation

$$\frac{\Xi, \Xi', Y{<:}!^m\Upsilon \vdash \Phi_X <: !\mathit{Top} \qquad \Xi, X{<:}\Phi_X, \Xi', Y{<:}!^m\Upsilon \vdash !^k\Psi'}{\Xi, \Xi', Y{<:}!^m\Upsilon \vdash !^k(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}\ .$$

Outer induction case: Let $\Theta' = \Theta'', Z{<:}\Phi_Z$ be a nonempty type context with $Z \neq Y$.

Inner base cases: Suppose $\Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z \vdash !^k\mathit{Top}$ has been derived using rule (*Top* type). By Lemma 4.18 and the definition of consistency we get proved type $\Theta, Y{<:}!^n\Phi_Y, \Theta'' \vdash \Phi_Z$. Outer induction then allows to conclude $\Theta, Y{<:}!^m\Upsilon, \Theta'' \vdash \Phi_Z$, leading to consistency of $\Theta, Y{<:}!^m\Upsilon, \Theta'', Z{<:}\Phi_Z$. Now we may apply (*Top* type) to obtain $\Theta, Y{<:}!^m\Upsilon, \Theta'', Z{<:}\Phi_Z \vdash !^k\mathit{Top}$. Rules (*Unit* type) and (*Qbit* type) may be handled similarly.

Assume $\Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z \vdash X$ with $X \neq Y$ has been derived by

$$\frac{\vdash \Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z}{\Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z \vdash X}\ \text{(linear type variable)}\ ,$$

where either $\Theta = \Xi, X{<:}\Phi_X, \Xi'$ or $\Theta'', Z{<:}\Phi_Z = \Xi, X{<:}\Phi_X, \Xi'$ for appropriate type contexts Ξ and Ξ'. As in the previous case, we may conclude $\Theta, Y{<:}!^m\Upsilon, \Theta'', Z{<:}\Phi_Z$ is consistent, and thus we may use rule (linear type variable) to obtain $\Theta, Y{<:}!^m\Upsilon, \Theta'', Z{<:}\Phi_Z \vdash X$.

Assume $\Theta, Y{<:}\Phi_Y, \Theta'', Z{<:}\Phi_Z \vdash Y$ has been derived using rule (linear type variable). As in the previous cases, we get consistency of $\Theta, Y{<:}!^m\Upsilon, \Theta'', Z{<:}\Phi_Z$ using outer induction. Thus, we can derive $\Theta, Y{<:}!^m\Upsilon, \Theta'', Z{<:}\Phi_Z \vdash Y$ using rule (linear type variable).

Suppose $\Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z \vdash !^kX$ with $X \neq Y$ has been derived by

$$\frac{\vdash \Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z}{\Theta, Y{<:}!^n\Phi_Y, \Theta'', Z{<:}\Phi_Z \vdash !^kX}\ \text{(nonlinear type variable)}\ ,$$

where either $\Theta = \Xi, X <: !\Phi_X, \Xi'$ or $\Theta'', Z <: \Phi_Z = \Xi, X <: !\Phi_X, \Xi'$ for appropriate type contexts Ξ and Ξ'. As in the previous cases, we get consistency of $\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z$ by outer induction, and thus we may use rule (nonlinear type variable) to obtain proved type $\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z \vdash !^k X$.

Assume $\Theta, Y <: !^n\Phi_Y, \Theta'', Z <: \Phi_Z \vdash !^k Y$ has been derived using rule (nonlinear type variable). Due to the single premise of (nonlinear type variable), we immediately know $n \geq 1$ and thus also $m \geq 1$, by premise $n > 0 \Rightarrow m > 0$ in $(\star)$. As in the previous cases, we get consistency of $\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z$ using outer induction. And in analogy to the respective case in the outer induction base, we then get to a derivation of $\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z \vdash !^k Y$.

Inner induction cases: Suppose $\Theta, Y <: !^n\Phi_Y, \Theta' \vdash \Psi$ has been derived by

$$\frac{\Theta, Y <: !^n\Phi_Y, \Theta'', Z <: \Phi_Z \vdash !^k\Phi' \qquad \Theta, Y <: !^n\Phi_Y, \Theta'', Z <: \Phi_Z \vdash !^k\Psi'}{\Theta, Y <: !^n\Phi_Y, \Theta'', Z <: \Phi_Z \vdash !^k(\Phi' \otimes \Psi')}\ \text{(product type)}\ .$$

Then, by inner induction, we derive the necessary premises (from the above ones, where consistency of $\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z$ is obtained by outer induction as already described in the inner base cases above) to enable derivation

$$\frac{\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z \vdash !^k\Phi' \qquad \Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z \vdash !^k\Psi'}{\Theta, Y <: !^m\Upsilon, \Theta'', Z <: \Phi_Z \vdash !^k(\Phi' \otimes \Psi')}\ \text{(product type)}\ .$$

Similar arguments cover the cases of rules (sum type) and (function type).
Assume $\Theta, Y <: !^n\Phi_Y, \Theta' \vdash \Psi$ has been derived by

$$\frac{\begin{array}{c}\vdash \Xi, \Xi', Y <: !^n\Phi_Y, \Theta' \\ \Xi, X <: \Phi_X, \Xi', Y <: !^n\Phi_Y, \Theta' \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')\end{array}}{\Xi, \Xi', Y <: !^n\Phi_Y, \Theta' \vdash (\forall X <: \Phi_X.\Psi')}\ \text{(linear-polymorphic type)}\ .$$

From the type context in the second premise it is clear that $X \neq Y$. We use inner induction (applied to the second premise) to obtain $\Xi, X <: \Phi_X, \Xi', Y <: !^m\Upsilon, \Theta' \vdash \Psi'$ and outer induction to ensure consistency of $\Xi, \Xi', Y <: !^m\Upsilon, \Theta'$. Having this, we can perform derivation

$$\frac{\begin{array}{c}\vdash \Xi, \Xi', Y <: !^m\Upsilon, \Theta' \\ \Xi, X <: \Phi_X, \Xi', Y <: !^m\Upsilon, \Theta' \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')\end{array}}{\Xi, \Xi', Y <: !^m\Upsilon, \Theta' \vdash (\forall X <: \Phi_X.\Psi')}\ \text{(linear-polymorphic type)}$$

to show that $\Theta, Y <: !^m\Upsilon, \Theta' \vdash (\forall X <: \Phi_X.\Psi')$ is a proved type.

The case where $\Theta, Y <: !^n\Phi_Y, \Theta' \vdash \Psi$ has been derived by

$$\frac{\begin{array}{c}\vdash \Theta, Y <: !^n\Phi_Y, \Xi, \Xi', Z <: \Phi_Z \\ \Theta, Y <: !^n\Phi_Y, \Xi, X <: \Phi_X, \Xi', Z <: \Phi_Z \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')\end{array}}{\Theta, Y <: !^n\Phi_Y, \Xi, \Xi', Z <: \Phi_Z \vdash (\forall X <: \Phi_X.\Psi')}\ \text{(linear-polymorphic type)}$$

is analogous.

Suppose $\Theta, Y <: !^n\Phi_Y, \Theta' \vdash \Psi$ has been derived by

$$\frac{\begin{array}{c}\Xi, \Xi', Y <: !^n\Phi_Y, \Theta' \vdash \Phi_X <: !\mathit{Top} \\ \Xi, X <: \Phi_X, \Xi', Y <: !^n\Phi_Y, \Theta' \vdash !^k\Psi'\end{array}}{\Xi, \Xi', Y <: !^n\Phi_Y, \Theta' \vdash !^k(\forall X <: \Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}\ .$$

Again, it is clear that $X \neq Y$. By inner induction (applied to the second premise) we get $\Xi, X <: \Phi_X, \Xi', Y <: !^m\Upsilon, \Theta' \vdash !^k\Psi'$. Applying Lemma 4.19 and outer induction to the first premise, we obtain consistency of $\Xi, \Xi', Y <: !^m\Upsilon, \Theta'$. By application of Lemma 4.18 to the second premise and by the definition of consistency for type contexts, we get proved type $\Xi \vdash \Phi_X$, which we weaken to $\Xi, \Xi', Y <: !^m\Upsilon, \Theta' \vdash \Phi_X$. Then we use rules ($\mathit{Top}$ supertype) and (! right) to derive proved subtype $\Xi, \Xi', Y <: !^m\Upsilon, \Theta' \vdash \Phi_X <: !\mathit{Top}$, where we get $\Phi_X = !\Phi'_X$ for some $\Phi'_X \in \mathcal{T}_{\mathit{type}}$ due to Corollary 4.17 applied to the first premise. Hence, we successfully collected all necessary premises to perform derivation

$$\frac{\Xi, \Xi', Y{<:}!^m\Upsilon, \Theta' \vdash \Phi_X <: !\mathit{Top} \quad \Xi, X{<:}\Phi_X, \Xi', Y{<:}!^m\Upsilon, \Theta' \vdash !^k\Psi'}{\Xi, \Xi', Y{<:}!^m\Upsilon, \Theta' \vdash !^k(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}$$

to obtain $\Theta, Y{<:}!^m\Upsilon, \Theta' \vdash !^k(\forall X{<:}\Phi_X.\Psi')$ as a proved type.

Finally, suppose $\Theta, Y{<:}!^n\Phi_Y, \Theta' \vdash \Psi$ has been derived by

$$\frac{\Theta, Y{<:}!^n\Phi_Y, \Xi, \Xi', Z{<:}\Phi_Z \vdash \Phi_X <: !\mathit{Top} \quad \Theta, Y{<:}!^n\Phi_Y, \Xi, X{<:}\Phi_X, \Xi', Z{<:}\Phi_Z \vdash !^k\Psi'}{\Theta, Y{<:}!^n\Phi_Y, \Xi, \Xi', Z{<:}\Phi_Z \vdash !^k(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}\ .$$

Again, we know $X \neq Y$. By inner induction (applied on the second premise) we get proved type $\Theta, Y{<:}!^m\Upsilon, \Xi, X{<:}\Phi_X, \Xi', Z{<:}\Phi_Z \vdash !^k\Psi'$. Applying Lemma 4.19 and outer induction to the first premise, we obtain consistency of $\Theta, Y{<:}!^m\Upsilon, \Xi, \Xi', Z{<:}\Phi_Z$. By application of Lemma 4.18 to the second premise and by the definition of consistency for type contexts, we get proved type $\Theta, Y{<:}!^n\Phi_Y, \Xi \vdash \Phi_X$ to which we apply outer induction to derive $\Theta, Y{<:}!^m\Upsilon, \Xi \vdash \Phi_X$, and then weaken this to obtain proved type $\Theta, Y{<:}!^m\Upsilon, \Xi, \Xi', Z{<:}\Phi_Z \vdash \Phi_X$. Afterwards we use rules (*Top* supertype) and (! right) to derive proved subtype $\Theta, Y{<:}!^m\Upsilon, \Xi, \Xi', Z{<:}\Phi_Z \vdash \Phi_X <: !\mathit{Top}$, where we get $\Phi_X = !\Phi'_X$ as before. Finally, we may perform derivation

$$\frac{\Theta, Y{<:}!^m\Upsilon, \Xi, \Xi', Z{<:}\Phi_Z \vdash \Phi_X <: !\mathit{Top} \quad \Theta, Y{<:}!^m\Upsilon, \Xi, X{<:}\Phi_X, \Xi', Z{<:}\Phi_Z \vdash !^k\Psi'}{\Theta, Y{<:}!^m\Upsilon, \Xi, \Xi', Z{<:}\Phi_Z \vdash !^k(\forall X{<:}\Phi_X.\Psi')}\ \text{(nonlinear-polymorphic type)}$$

to obtain proved type $\Theta, Y{<:}!^m\Upsilon, \Theta' \vdash !^k(\forall X{<:}\Phi_X.\Psi')$.

□

Now that we have got an insight into the importance of leading exponentials for type bounds in type contexts, we next investigate the importance of their number. What we discover is in fact neither new nor very surprising. In the beginning of the treatment of our polymorphic type system, we have emphasized the conceptual existence of type isomorphism $!\Phi \cong !!\Phi$ for any type term Φ.[33]

But there is another aspect to discover (which is also nothing completely new). Consider a linear type term Φ'. When we introduced subtyping in the beginning of the current section, we said an object of type $!\Phi'$ is of type Φ' and shows the additional property of being duplicable. What we find in the next lemma as well, is that for any type term $!^n\Phi$ which is derivable with leading exponentials (equipped with an appropriate type context, of course), we can also derive Φ without leading exponentials (using the same type context).

Lemma 4.31. *Let $m, n \geq 0$. If $\Theta \vdash !^{n+1}\Phi$ can be derived as a proved type, then we can also derive $\Theta \vdash !^m\Phi$.*

Proof. We show this by induction on the derivation of $\Theta \vdash !^{n+1}\Phi$:

Base cases: Let $\Theta \vdash !^{n+1}\mathit{Top}$ be derived by rule (*Top* type). Of course, we can also derive $\Theta \vdash !^m\mathit{Top}$ for any $m \geq 0$. The same holds for rule (*Unit* type). The case of $\Theta \vdash !^{n+1}\mathit{Qbit}$ is different, however, since it is not derivable using rule (*Qbit* type) or any other rule.

We do not need to consider rule (linear type variable), since it is not possible to derive any proved type $\Theta \vdash !^{n+1}X$ using this rule.

Suppose $\Xi, X{<:}!\Phi_X, \Xi' \vdash !^{n+1}X$ has been derived using rule (nonlinear type variable). But then, we can also perform derivation

$$\frac{\vdash \Xi, X{<:}!\Phi_X, \Xi'}{\Xi, X{<:}!\Phi_X, \Xi' \vdash !^m X}\ \text{(nonlinear type variable)}\ .$$

Let $\Theta \vdash !^{n+1}(\Phi' \multimap \Psi')$ be derived by rule (function type). Using the same premises $\Theta \vdash \Phi'$ and $\Theta \vdash \Psi'$, we may also derive $\Theta \vdash !^m(\Phi' \multimap \Psi')$ using the same rule.

[33] We say the mentioned isomorphism exists *conceptually*, since we never formally introduced it and do not intend to do so within the present work.

Induction cases: Suppose $\Theta \vdash !^{n+1}\Phi$ has been derived by

$$\frac{\Theta \vdash !^{n+1}\Phi' \qquad \Theta \vdash !^{n+1}\Psi'}{\Theta \vdash !^{n+1}(\Phi' \otimes \Psi')} \text{ (product type) .}$$

Then induction tells us $\Theta \vdash !^m\Phi'$ and $\Theta \vdash !^m\Psi'$ are derivable proved types, and thus we can use rule (product type) to derive $\Theta \vdash !^m(\Phi' \otimes \Psi')$. The same straightforward argument applies to rule (sum type).

We do not have to consider rule (linear-polymorphic type), since it is impossible to derive any proved type $\Theta \vdash !^{n+1}\Phi$ using this rule.

Finally suppose $\Theta \vdash !^{n+1}\Phi$ has been derived by

$$\frac{\Xi, \Xi' \vdash \Phi_X <: !Top \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash !^{n+1}\Psi'}{\Xi, \Xi' \vdash !^{n+1}(\forall X{<:}\Phi_X.\Psi')} \text{ (nonlinear-polymorphic type)}$$

with $\Xi, \Xi' = \Theta$. By induction, we immediately get proved type $\Xi, X{<:}\Phi_X, \Xi' \vdash !^m\Psi'$. And thus, we can easily perform the following derivation to obtain $\Xi, \Xi' \vdash !^m(\forall X{<:}\Phi_X.\Psi')$:

$$\frac{\Xi, \Xi' \vdash \Phi_X <: !Top \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash !^m\Psi'}{\Xi, \Xi' \vdash !^m(\forall X{<:}\Phi_X.\Psi')} \text{ (nonlinear-polymorphic type) .}$$

□

We put the results from the above Lemma 4.31 into two new derivation rules, so that we can later clearly distinguish between cases where we introduce more leading exponentials into a proved type (in which at least one leading exponential already exists), or where we eliminate leading exponentials from a proved type.

Corollary 4.32. *We can derive the following rules for $m, n \geq 0$:*

$$\frac{\Theta \vdash !\Phi}{\Theta \vdash !^{m+1}\Phi} \text{ (! amplification) ,} \qquad \frac{\Theta \vdash !^{n+1}\Phi}{\Theta \vdash \Phi} \text{ (! elimination) .}$$

Proof. These rules are immediate consequences of Lemma 4.31:

- To obtain rule (! amplification), we apply Lemma 4.31, where we set $n = 0$ and $m \geq 1$ in the lemma.
- To come up with rule (! elimination), we apply Lemma 4.31, where we set $n \geq 1$ and $m = 0$ in the lemma.

□

This last two derivation rules together with all preceding results yield a quite powerful collection of useful tools. We will make extensive use of that handy toolbox in the subsequent sections to tackle more advanced topics and aspects of the type system we develop.

4.2.3 Enforceable linearity and strictly linear type terms

In this subsection we make sure our system exhibits a key feature that we want it to have, namely that type constant *Qbit* cannot be maneuvered into a "dangerous spot" during derivation of proved types. What we try to achieve by this is that when a function term contains quantum data and is assigned a type, the type system shall guarantee that this function term is not regarded as duplicable, since any attempt to duplicate such a function term (by means of a general cloning operation) would need operations that are physically impossible, as Theorem 2.5 (no-cloning theorem) states. We already elaborated extensively on this matter when we have been discussing the implications of the no-cloning theorem for *QLC* function terms and when we have been motivating our polymorphic type system in the very beginning of the current section.

As we have already mentioned several times, we want to prohibit certain type terms such as $!Qbit$ and $!(Qbit \otimes \dots)$ and the like to become part of a proved type. And since just claiming our system cannot derive proved types which include such type terms does not mean any certainty, we formalize the idea of type terms that shall only occur in linear form as subexpressions of proved types.

For this venture, we first need to pin down exactly, which type terms shall be restricted to only occur linearly.

Definition 4.33 (strictly linear type terms).
We inductively define the set $\mathcal{T}_{sl}$ of *strictly linear type terms* as follows:

$$\begin{aligned}
& & Qbit \in \mathcal{T}_{sl} \\
(\Phi \otimes \Psi) \in \mathcal{T}_{type} \ & \wedge \big(\Phi \in \mathcal{T}_{sl} \vee \Psi \in \mathcal{T}_{sl}\big) & \Longrightarrow (\Phi \otimes \Psi) \in \mathcal{T}_{sl} \\
(\Phi \oplus \Psi) \in \mathcal{T}_{type} \ & \wedge \big(\Phi \in \mathcal{T}_{sl} \vee \Psi \in \mathcal{T}_{sl}\big) & \Longrightarrow (\Phi \oplus \Psi) \in \mathcal{T}_{sl} \\
(\forall X{<:}\Phi_X.\Psi) \in \mathcal{T}_{type} \ & \wedge \big(\big(\Phi_X \in \mathcal{T}_{sl} \wedge X \in \mathit{nftyv}(!\Psi)\big) \vee \Psi \in \mathcal{T}_{sl}\big) & \Longrightarrow (\forall X{<:}\Phi_X.\Psi) \in \mathcal{T}_{sl}
\end{aligned}$$

Let us take a short look at a few examples:

- $\Upsilon_1 = (Qbit \otimes !(Qbit \multimap Qbit))$ and $\Upsilon_2 = (\forall X{<:}(Qbit \otimes !Top).X)$
  are a strictly linear type terms for which we can derive proved types $\vdash \Upsilon_1$ and $\vdash \Upsilon_2$;
- $\Upsilon_3 = (!Qbit \otimes Qbit)$
  is strictly linear, but there is no type context Θ so that we could derive $\Theta \vdash \Upsilon_3$;
- $\Upsilon_4 = (\forall X{<:}Top.(!X \multimap Qbit))$
  is *not* strictly linear, but there is also no proved type $\Theta \vdash \Upsilon_4$.[34]

The first notable and quite obvious fact about strictly linear type terms is $\mathcal{T}_{sl} \subset \mathcal{T}_{type}$. But this is indeed not very surprising, since it only makes sense to exclude type terms that have actually been there from the beginning.

The second notable (and much more interesting) fact is the strong connection with nonlinear occurrences of free type variables in type terms (cf. Definition 4.5). This is by no means a coincidence. When we have been discussing the notion of nonlinear free type variables earlier, we claimed these were marking "dangerous spots" in type terms, where no type term such as $Qbit$ shall be put by substitution. In the above definition of strictly linear type terms these "dangerous spots" are exactly filled with type terms that shall stay linear. Dangerous in this respect means to be affected by an exponential. And since a leading exponential in front of $(\Phi \otimes \Psi)$ affects both Φ and Ψ, $(\Phi \otimes \Psi)$ shall stay linear if Φ or Ψ has to. (Recall that we have said earlier – when discussing the type derivation rules – $(\Phi \otimes \Psi)$ shall be duplicable if both Φ and Ψ are.) This has been the reason why in function call $\mathit{nftyv}(!^n(\Phi \otimes \Psi)) = \mathit{nftyv}(!^n\Phi) \cup \mathit{nftyv}(!^n\Psi)$ the n leading exponentials are propagated. And this is also the reason why $(\Phi \otimes \Psi)$ is strictly linear, if Φ or Ψ is strictly linear. Hence, the notions of nonlinear free type variables and strictly linear type terms go back to the same idea. To illustrate this connection even further and explain the shape of the last defining line of set $\mathcal{T}_{sl}$ in Definition 4.33, consider a strictly linear type term $(\forall X{<:}\Phi_X.\Psi) \in \mathcal{T}_{sl}$. It became strictly linear because of Ψ being strictly linear or because of X occurring as nonlinear free type variable in $!\Psi$ and Φ_X being strictly linear. This exactly corresponds to the application of function nftyv to $!(\forall X{<:}\Phi_X.\Psi)$, i.e.

$$\mathit{nftyv}(!(\forall X{<:}\Phi_X.\Psi)) = \mathit{nftyv}(!^k\Phi_X) \cup \big(\mathit{nftyv}(!\Psi) \setminus \{X\}\big) \text{ , where } k := \begin{cases} 1, \text{ if } X \in \mathit{nftyv}(!\Psi), \\ 0, \text{ otherwise.} \end{cases}$$

There a leading exponential is propagated to function application $\mathit{nftyv}(!^k\Phi_X)$ only if $X \in \mathit{nftyv}(!\Psi)$ is fulfilled. This keeps Y from being classified as nonlinear in the example type term $!(\forall X{<:}Y.(X \multimap X))$, for instance. On the other hand, $(\forall X{<:}Qbit.(X \multimap X))$ is not strictly linear. This is in accordance with our understanding that a function of type $!(Qbit \multimap Qbit)$ (which results for example from applying type term $Qbit$ to a function term of type $!(\forall X{<:}Qbit.(X \multimap X))$, as we will see later) may be very well used multiple times in a function term even if the argument it works on and the result it produces are of linear type. And in full correspondence to this we can derive $\vdash !(Qbit \multimap Qbit)$ by

[34] Type term Υ_4 belongs to a class of type terms for which we cannot derive proved types, but it still does not contain a subexpression that is a nonlinear strictly linear type term such as $!Qbit$. The danger of Υ_4 lies in "type applications" of the form $((\forall X{<:}Top(!X \multimap Qbit))\ Qbit) \rightsquigarrow (!X \multimap Qbit)[Qbit/X]$. We shortly discuss this class of type terms at the end of the current subsection 4.2.3.

$$\dfrac{\dfrac{\vdash \emptyset}{\vdash \mathit{Qbit}}\,(\mathit{Qbit}\text{ type}) \qquad \dfrac{\vdash \emptyset}{\vdash \mathit{Qbit}}\,(\mathit{Qbit}\text{ type})}{\vdash !(\mathit{Qbit} \multimap \mathit{Qbit})}\,(\text{function type})\ .$$

The third notable fact about strictly linear type terms hints in the same direction as the previous one does in that it also emphasizes the connection between strictly linear type terms and occurrences of nonlinear free type variables. It has, however, a slightly different flavor. Obviously, each strictly linear type term has at least one occurrence of type constant *Qbit*. This is so, since the definition of strictly linear type terms is of an inductive nature and the only base case is $\mathit{Qbit} \in \mathcal{T}_{sl}$. Now consider a strictly linear type term Φ and a type variable $Z \in \mathcal{V}_{type}$, where we assume Z does not occur in Φ (neither free nor bound) and Z is not bound by a type abstraction in Φ. Let us write $\Phi_{\mathit{Qbit}\mapsto Z}$ for the type term which results from Φ by replacing all occurrences of *Qbit* with Z. Recall examples Υ_1 and Υ_3 from above. For these, we get $\Upsilon_{1,\mathit{Qbit}\mapsto Z} = (Z \otimes !(Z \multimap Z))$ and $\Upsilon_{3,\mathit{Qbit}\mapsto Z} = (!Z \otimes Z)$. As type terms $\Upsilon_{1,\mathit{Qbit}\mapsto Z}$ and $\Upsilon_{3,\mathit{Qbit}\mapsto Z}$ show, Z might or might not occur nonlinearly in $\Phi_{\mathit{Qbit}\mapsto Z}$ for arbitrary strictly linear Φ, but Z surely is a nonlinear free type variable in $!\Upsilon_{1,\mathit{Qbit}\mapsto Z}$ and $!\Upsilon_{3,\mathit{Qbit}\mapsto Z}$, as we can easily check.

The following lemma confirms the just made observation for the general case $!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z}$ for arbitrary strictly linear type terms Φ and $n \geq 0$.

Lemma 4.34. *If Φ is a strictly linear type term, then Z appears as nonlinear free type variable in $!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z}$ with $n \geq 0$. In other words, $\Phi \in \mathcal{T}_{sl}$ implies $Z \in \mathit{nftyv}\big(!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z}\big)$.*

Proof. We show this by induction on the structure of $\Phi \in \mathcal{T}_{sl}$.

Base case: Let Φ be of the form $\Phi = \mathit{Qbit}$. Then

$$\mathit{nftyv}(!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z}) = \mathit{nftyv}(!^{n+1}Z) = \{Z\}\ .$$

Induction case: Let Φ be of the form $\Phi = (\Phi' \otimes \Psi')$. Since Φ is strictly linear, we know $\Phi' \in \mathcal{T}_{sl}$ or $\Psi' \in \mathcal{T}_{sl}$. But then, we have $Z \in \mathit{nftyv}(!^{n+1}\Phi'_{\mathit{Qbit}\mapsto Z}) \cup \mathit{nftyv}(!^{n+1}\Psi'_{\mathit{Qbit}\mapsto Z}) = \mathit{nftyv}(!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z})$ by induction.

Analogously, we have $Z \in \mathit{nftyv}(!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z})$ for the case $\Phi = (\Phi' \oplus \Psi')$.

Assume Φ has the form $\Phi = (\forall X{<:}\Phi'.\Psi')$, where we assume $X \neq Z$ without loss of generality (since we identify α-equivalent type terms). Then we know $\Psi' \in \mathcal{T}_{sl}$ holds, or $\Phi' \in \mathcal{T}_{sl}$ and $X \in \mathit{nftyv}(!\Psi')$ is fulfilled. If the first part $\Psi' \in \mathcal{T}_{sl}$ is true, then we know $Z \in \mathit{nftyv}(!^{n+1}\Psi'_{\mathit{Qbit}\mapsto Z})$ by induction. If the second part is true, induction yields $Z \in \mathit{nftyv}(!^{n+1}\Phi'_{\mathit{Qbit}\mapsto Z})$. By inspection of the definition of function *nftyv*, we see $\mathit{nftyv}(!^{n+1}\Phi'_{\mathit{Qbit}\mapsto Z}) = \mathit{nftyv}(!\Phi'_{\mathit{Qbit}\mapsto Z})$, i.e. regarding the outcome it does not make a difference whether we have one leading exponential in front of $\Phi'_{\mathit{Qbit}\mapsto Z}$ or more than one. Moreover, we get $\mathit{nftyv}(!\Psi') \subseteq$
$\mathit{nftyv}(!\Psi'_{\mathit{Qbit}\mapsto Z})$, since Ψ' and $\Psi'_{\mathit{Qbit}\mapsto Z}$ differ only in the occurrences of *Qbit* and Z.
Hence, in total we have $Z \in \mathit{nftyv}(!^{n+1}\Psi'_{\mathit{Qbit}\mapsto Z})$ or $Z \in \mathit{nftyv}(!\Phi'_{\mathit{Qbit}\mapsto Z})$ with
$X \in \mathit{nftyv}(!^{n+1}\Psi'_{\mathit{Qbit}\mapsto Z})$, which corresponds to

$$\begin{aligned} Z \in \mathit{nftyv}(!^{n+1}\Phi_{\mathit{Qbit}\mapsto Z}) &= \mathit{nftyv}\big(!^{n+1}(\forall X{<:}\Phi'_{\mathit{Qbit}\mapsto Z}.\Psi'_{\mathit{Qbit}\mapsto Z})\big) \\ &= \mathit{nftyv}(!^{k}\Phi'_{\mathit{Qbit}\mapsto Z}) \cup \big(\mathit{nftyv}(!^{n+1}\Psi'_{\mathit{Qbit}\mapsto Z}) \setminus \{X\}\big)\ , \\ &\qquad \text{where } k := \begin{cases} 1, \text{ if } X \in \mathit{nftyv}(!^{n+1}\Psi'_{\mathit{Qbit}\mapsto Z}), \\ 0, \text{ otherwise.} \end{cases} \end{aligned}$$

□

This result reveals once more the strong connection between the definitions of function *nftyv* and the set $\mathcal{T}_{sl}$. With its help we can switch our focus to occurrences of nonlinear free type variables instead of strictly linear type terms, when we obtain our first result stating that particular type terms cannot be part of a proved type in the next lemma.

But before we formulate Lemma 4.35, let us take a look at the core idea underlying its proof. We use the following observations, parts of which we have already explained above. On the one hand, nonlinear occurrences of type variables are directly or indirectly affected by exponentials. This is reflected in the definition of function *nftyv* and its mechanisms of propagation of exponentials. We have already discussed

this in several places. On the other hand, we claim that we cannot derive proved types $\Theta \vdash \Phi$, where type constant *Qbit* occurs in a place in Φ where it is affected by an exponential. These two observations clearly lead to the conclusion that if *Qbit* is substituted for a nonlinear X in a type term Ψ, then *Qbit* will get affected by exponentials, and thus we may not derive $\Theta' \vdash \Psi[Qbit/X]$ anymore, even if $\Theta' \vdash \Psi$ has been derivable before.

To facilitate a proof of those claims, we need the formalize the just explained ideas. Our focus shall be on type variables, type constants and exponentials that affect them. The way of how an exponential can affect a type variable is made clear in the definition of function *nftyv*.

At first, we need to introduce a way to address occurrences of type constants and type variables in type terms. We do this by assigning a position to each individual occurrence of type constants and type variables. As positions we use words over the alphabet $\{1,2\}$, where ε denotes the empty word. Function $\pi : \mathcal{T}_{type} \to \mathcal{P}\big(\{1,2\}^* \times \big(\{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\} \cup \mathcal{V}_{type}\big)\big)$ generates pairs of occurrences of type constants and type variables in a type term together with their respective positions as follows. For all $C \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\}$, $X \in \mathcal{V}_{type}$ and $\Phi, \Psi \in \mathcal{T}_{type}$, we define

$$\begin{aligned}
\pi(C) &:= \big\{(\varepsilon, C)\big\} \;, \\
\pi(X) &:= \big\{(\varepsilon, X)\big\} \;, \\
\pi\big((\Phi \otimes \Psi)\big) &= \pi\big((\Phi \oplus \Psi)\big) = \pi\big((\Phi \multimap \Psi)\big) = \\
\pi\big((\forall X{<}{:}\Phi.\Psi)\big) &:= \big\{(1p, S) \;\big|\; (p, S) \in \pi(\Phi)\big\} \;\cup\; \big\{(2p, S) \;\big|\; (p, S) \in \pi(\Psi)\big\} \;, \\
\pi(!\Phi) &:= \pi(\Phi) \;.
\end{aligned}$$

Let us take a look at an example:

$$\begin{aligned}
&\pi\big(\,!!((Y \otimes !\mathit{Unit}) \oplus (\forall X{<}{:}!\mathit{Top}.!((X \otimes \mathit{Qbit}) \multimap !\mathit{Unit})))\,\big) \\
&\qquad = \big\{(11, Y), (12, \mathit{Unit}), (21, \mathit{Top}), (2211, X), (2212, \mathit{Qbit}), (222, \mathit{Unit})\big\} \;.
\end{aligned}$$

In fact, the positions are not of importance to us but are necessary to distinguish different occurrences of one type constant or type variable.

Function π ignores occurring exponentials completely. But what happens if we connect the idea of collecting all occurring type constants and type variables and of counting the number of exponentials that affect them, and do both during a recursive traversal of the whole type term? Clearly, function *nftyv* does something similar, but does not count exponentials, but rather propagates them. Thus, we can also use propagation and then count as soon as we (almost) arrived at a type constant or type variable. What would be the outcome in the above example? When we do this counting by hand, we come to the result

$$!!((\,{}_2Y \otimes !\,{}_3\mathit{Unit}) \oplus (\forall X{<}{:}!\,{}_1\mathit{Top}.!((\,{}_0X \otimes \,{}_0\mathit{Qbit}) \multimap !\,{}_1\mathit{Unit}))) \;,$$

where the small number in front of each type constant and type variable is the sought number of exponentials affecting it.

In the proof of Lemma 4.35 we define function $\widehat{\pi}$, which is a combination of functions π and *nftyv* and does the counting for us. The result for our example is the following set of triples, where the counted numbers are written in the third component:

$$\begin{aligned}
&\widehat{\pi}\big(\,!!((Y \otimes !\mathit{Unit}) \oplus (\forall X{<}{:}!\mathit{Top}.!((X \otimes \mathit{Qbit}) \multimap !\mathit{Unit})))\,\big) \\
&\qquad = \big\{(11, Y, 2), (12, \mathit{Unit}, 3), (21, \mathit{Top}, 1), (2211, X, 0), (2212, \mathit{Qbit}, 0), (222, \mathit{Unit}, 1)\big\} \;.
\end{aligned}$$

Function $\widehat{\pi}$ holds the key for a proof by contradiction. We use it to make three important but indeed not very surprising observations on occurrences of type constants and type variables in proved types:

- firstly, *nonlinear occurrences* of free type variables are affected by *at least one* exponential,
- secondly, occurrences of type constant *Qbit* are *never* affected by exponentials,
- thirdly, when free type variable X has one occurrence at position p in type term Ψ at which it is affected by m exponentials, then type constant *Qbit* at position p in $\Psi[Qbit/X]$ is also affected by m exponentials.

Clearly, this would lead to a contradiction if $\Theta \vdash \Psi[\mathit{Qbit}/X]$ were derivable for a nonlinear free type variable X in Ψ. Since then, on the one hand, *Qbit* must not be affected by exponentials in any of its occurrences in $\Psi[\mathit{Qbit}/X]$. On the other hand, there must be at least one occurrence of *Qbit* in $\Psi[\mathit{Qbit}/X]$ that is affected by at least one exponential.

All these considerations culminate in the following result.

Lemma 4.35. *Let $\Theta \vdash \Psi$ be an arbitrary proved type and let $X \in \mathcal{V}_{type}$ be a type variable. If $X \in \mathit{nftyv}(\Psi)$, then $\Theta' \vdash \Psi[\mathit{Qbit}/X]$ cannot be derived as a proved type for any type context Θ'.*

Proof. We define a modified version of the above described function π, namely

$$\widehat{\pi} : \mathcal{T}_{type} \to \mathcal{P}\big(\{1,2\}^* \times \big(\{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\} \cup \mathcal{V}_{type}\big) \times \mathbb{N}\big) \ .$$

We can think about $\widehat{\pi}$ as a function computing for each position in a type term Φ an "amount of nonlinearity" affecting it, expressed by a non-negative integer $m \geq 0$:

$$\begin{aligned}
\widehat{\pi}(!^n C) &:= \big\{(\varepsilon, C, n)\big\} \text{ for } C \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\} \ , \\
\widehat{\pi}(!^n X) &:= \big\{(\varepsilon, X, n)\big\} \text{ for } X \in \mathcal{V}_{type} \ , \\
\widehat{\pi}\big(!^n(\Phi \otimes \Psi)\big) &= \\
\widehat{\pi}\big(!^n(\Phi \oplus \Psi)\big) &:= \big\{(1p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(!^n\Phi)\big\} \cup \big\{(2p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(!^n\Psi)\big\} \ , \\
\widehat{\pi}\big(!^n(\Phi \multimap \Psi)\big) &:= \big\{(1p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(\Phi)\big\} \quad \cup \big\{(2p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(\Psi)\big\} \ , \\
\widehat{\pi}\big(!^n(\forall X{<:}\Phi_X.\Psi)\big) &:= \big\{(1p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(!^k\Phi_X)\big\} \cup \big\{(2p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(!^n\Psi)\big\} \ , \\
&\qquad \text{where } k := \begin{cases} 1, \text{ if } X \in \mathit{nftyv}(!^n\Psi), \\ 0, \text{ otherwise.} \end{cases}
\end{aligned}$$

Obviously and indeed not much of a surprise, function $\widehat{\pi}$ shows a strong similarity to function *nftyv*. The most important fact is that it has the same behavior regarding the propagation of leading exponentials.

We now show by induction that for each proved type $\Theta \vdash \Phi$ the following properties hold:

(i) for all type variables $X \in \mathit{nftyv}(\Phi)$ there exists $(p, X, m) \in \widehat{\pi}(\Phi)$ such that $m \geq 1$, and

(ii) for all $(p, \mathit{Qbit}, m) \in \widehat{\pi}(\Phi)$ it holds $m = 0$.

Base cases: Assume $\Theta \vdash !^n \mathit{Top}$ has been derived using rule (*Top* type). Then (i) and (ii) trivially hold, since $!^n \mathit{Top}$ neither contains a nonlinear free type variable, nor does it have a position occupied by *Qbit*. The same holds for a type term $\Theta \vdash !^n \mathit{Unit}$ derived using (*Unit* type).

Assume $\Theta \vdash \mathit{Qbit}$ has been derived using rule (*Qbit* type). Property (i) holds, since there is no nonlinear free type variable in type term *Qbit*. On the other hand, $\widehat{\pi}(\mathit{Qbit}) = \{(\varepsilon, \mathit{Qbit}, 0)\}$ shows that (ii) also holds in this case.

Suppose $\Theta, X{<:}\Phi_X, \Theta' \vdash X$ has been derived with rule (linear type variable). Then (ii) certainly holds, since there is no (p, Qbit, m) in $\widehat{\pi}(X)$. Regarding property (i), we know it trivially holds in this case, because $\mathit{nftyv}(X) = \emptyset$.

Next, suppose $\Theta, X{<:}\Phi_X, \Theta' \vdash !^n X$ has been derived using rule (nonlinear type variable). As in the previous case, (ii) clearly holds. Regarding property (i), we know it trivially holds in case of $n = 0$ since $\mathit{nftyv}(!^0 X) = \emptyset$. If $n \geq 1$, we have $\widehat{\pi}(!^n X) = \{(\varepsilon, X, n)\}$. But now, it is obvious that (i) holds.

Induction cases: Assume $\Theta \vdash !^n(\Phi' \multimap \Psi')$ has been derived using rule (function type). Then

$$\widehat{\pi}\big(!^n(\Phi' \multimap \Psi')\big) = \underbrace{\big\{(1p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(\Phi')\big\}}_{=:\ \pi_1} \cup \underbrace{\big\{(2p, S, m) \bigm| (p, S, m) \in \widehat{\pi}(\Psi')\big\}}_{=:\ \pi_2} \ .$$

By induction we know (i) and (ii) hold for $\Theta \vdash \Phi'$ and $\Theta \vdash \Psi'$, respectively, since these are the premises of rule (function type).

(i): Since $\mathit{nftyv}(!^n(\Phi' \multimap \Psi')) = \mathit{nftyv}(\Phi') \cup \mathit{nftyv}(\Psi')$, we have $X \in \mathit{nftyv}(\Phi')$ or $X \in \mathit{nftyv}(\Psi')$ for each $X \in \mathit{nftyv}(!^n(\Phi' \multimap \Psi'))$. Then there exists a $(p, X, m) \in \widehat{\pi}(\Phi')$ and thus a $(1p, X, m) \in \pi_1$ with $m \geq 1$, if $X \in \mathit{nftyv}(\Phi')$, and there exists a $(p', X, m') \in \widehat{\pi}(\Psi')$ and thus a $(2p', X, m') \in \pi_2$ with $m' \geq 1$, if $X \in \mathit{nftyv}(\Psi')$. But this means that (i) also holds for $\Theta \vdash !^n(\Phi' \multimap \Psi')$.

(ii): Because of $m_1 = 0$ for all $(p, Qbit, m_1) \in \widehat{\pi}(\Phi')$ (by induction), and due to the way π_1 is constructed from $\widehat{\pi}(\Phi')$, we also have $m_1 = 0$ for all $(1p, Qbit, m_1) \in \pi_1$. Obviously, an analogous fact holds for π_2, $\widehat{\pi}(\Psi')$ and their elements. Hence, we have $m = 0$ for all $(p, Qbit, m) \in \widehat{\pi}(!^n(\Phi' \multimap \Psi')) = \pi_1 \cup \pi_2$.

We may argue along similar lines for the cases of rules (product type) and (sum type).

Now suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi')} \text{ (linear-polymorphic type)}$$

with $\Xi, \Xi' = \Theta$. Then

$$\widehat{\pi}\big((\forall X{<:}\Phi_X.\Psi')\big) = \big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^k\Phi_X)\big\} \cup \big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(\Psi')\big\} \;,$$

where k is defined as above, here for $n = 0$. But since the third premise states $X \notin \mathit{nftyv}(\Psi')$, we immediately know $k = 0$. The second premise assumes $\Xi, X{<:}\Phi_X, \Xi' \vdash \Psi'$ to be a proved type. From Lemma 4.18 and the definition of consistency for type contexts we conclude $\Xi \vdash \Phi_X$ is a proved type. By induction we then know (i) and (ii) hold for $\Xi \vdash \Phi_X$ and $\Xi, X{<:}\Phi_X, \Xi' \vdash \Psi'$, respectively.

(i): By definition of function *nftyv* we have $\mathit{nftyv}((\forall X{<:}\Phi_X.\Psi')) = \mathit{nftyv}(!^0\Phi_X) \cup \mathit{nftyv}(\Psi')$, since we already concluded $k = 0$ above. But now we can use similar arguments as in the case of rule (function type) to establish property (i) for the current case.

(ii): Again, we may reuse the respective arguments of case (function type) in an analogous way for property (ii) in the current case.

Finally, assume $\Theta \vdash \Phi$ has been derived by

$$\frac{\Xi, \Xi' \vdash \Phi_X <: !\mathit{Top} \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash !^n\Psi'}{\Xi, \Xi' \vdash !^n(\forall X{<:}\Phi_X.\Psi')} \text{ (nonlinear-polymorphic type)} \;,$$

again with $\Xi, \Xi' = \Theta$. Then

$$\widehat{\pi}\big(!^n(\forall X{<:}\Phi_X.\Psi')\big) = \big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^k\Phi_X)\big\} \cup \big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n\Psi')\big\} \;,$$

where k is defined as above. Premise $\Xi, \Xi' \vdash \Phi_X <: !\mathit{Top}$ and Corollary 4.17 tell us that Φ_X is of the form $!\Phi'_X$ for some type term Φ'_X. And analogous to the previous case, we may conclude $\Xi \vdash \Phi_X$. Induction then tells us that properties (i) and (ii) also hold for proved types $\Xi \vdash !\Phi'_X$ and $\Xi, X{<:}!\Phi'_X, \Xi' \vdash !^n\Psi'$.

(i): We have already argued the case of $k = 0$ in a very similar situation in the case of (linear-polymorphic type).

Let $k = 1$. For now, we use a result which we show a bit later, namely $\widehat{\pi}(!!\Phi'') \succeq \widehat{\pi}(!\Phi'')$, which we define as

$$\widehat{\pi}(\Upsilon) \succeq \widehat{\pi}(\Upsilon') \;\text{ if and only if }\; \forall (p, S, m) \in \widehat{\pi}(\Upsilon).\; \exists (p, S, m') \in \widehat{\pi}(\Upsilon').\; m \geq m' \;.$$

This allows us to immediately deduce from (i) holds for $\Xi \vdash !\Phi'_X$ (due to induction) that (i) also holds for proved type $\Xi \vdash !!\Phi'_X$. This leaves us in a position where we may apply similar reasoning as in the case of property (i) for rule (function type).

(ii): We again use a result here that we prove soon: $\widehat{\pi}(!!\Phi'_X) \subseteq_{Qbit} \widehat{\pi}(!\Phi'_X)$, where $\widehat{\pi}(\Upsilon) \subseteq_{Qbit} \widehat{\pi}(\Upsilon')$ holds if and only if all $(p, Qbit, m) \in \widehat{\pi}(\Upsilon)$ are also in $\widehat{\pi}(\Upsilon')$. Having this, we can extend the fact that (ii) holds for $\Xi \vdash !\Phi'_X$ to proved type $\Xi \vdash !!\Phi'_X$. Thus, we may argue along the same lines as in case (function type) to establish (ii) for $\Xi, \Xi' \vdash !^n(\forall X{<:}!\Phi'_X.\Psi')$.

$\Diamond$

Now we have finished the induction. However, we left two properties open to show (for all proved types $\Theta \vdash !\Phi$):

(i') $\widehat{\pi}(!!\Phi) \succeq \widehat{\pi}(!\Phi)$, and

(ii') $\widehat{\pi}(!!\Phi) \subseteq_{Qbit} \widehat{\pi}(!\Phi)$.

Since both properties are directly defined on function $\widehat{\pi}$, we prove them by induction on the structure of type term Φ as it appears in the definition of $\widehat{\pi}(\Phi)$:

Base cases: Let Φ be of the form $!^n S$, where $S \in \{Top, Unit, Qbit\} \cup \mathcal{V}_{type}$.

(i'): By definition we have $\widehat{\pi}(!!!^n S) = \{(\varepsilon, S, n+2)\}$ and $\widehat{\pi}(!!^n S) = \{(\varepsilon, S, n+1)\}$. Clearly, it holds $\widehat{\pi}(!!!^n S) \succeq \widehat{\pi}(!!^n S)$.[35]

(ii'): Inspecting the rules for derivation of proved types, we see $\Theta \vdash !^{n'} Qbit$ can only be derived for $n' = 0$. Hence, there neither is a triple $(p, Qbit, m)$ in $\widehat{\pi}(!!!^n S)$ nor in $\widehat{\pi}(!!^n S)$ for any $m \in \mathbb{N}$. Thus $\widehat{\pi}(!!!^n S) \subseteq_{Qbit} \widehat{\pi}(!!^n S)$ trivially holds.

Induction cases: Let Φ be of the form $!^n(\Phi' \otimes \Psi')$. By definition we have

$$\widehat{\pi}\big(!^l!^n(\Phi' \otimes \Psi')\big) \;=\; \underbrace{\big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^l!^n\Phi')\big\}}_{=:\ \pi_3} \cup \underbrace{\big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^l!^n\Psi')\big\}}_{=:\ \pi_4}$$

for $l \in \{1, 2\}$.

(i'): Induction yields $\widehat{\pi}(!!!^n\Phi') \succeq \widehat{\pi}(!!^n\Phi')$ and $\widehat{\pi}(!!!^n\Psi') \succeq \widehat{\pi}(!!^n\Psi')$. But then we may conclude from the above definition of $\widehat{\pi}(!^l!^n(\Phi' \otimes \Psi'))$ and the construction of π_3 and π_4 that $\widehat{\pi}(!!!^n(\Phi' \otimes \Psi')) \succeq \widehat{\pi}(!!^n(\Phi' \otimes \Psi'))$ also holds.

(ii'): Here, we use induction to conclude $\widehat{\pi}(!!!^n\Phi') \subseteq_{Qbit} \widehat{\pi}(!!^n\Phi')$ and $\widehat{\pi}(!!!^n\Psi') \subseteq_{Qbit} \widehat{\pi}(!!^n\Psi')$. Again, the way of how π_3 and π_4 are constructed and the definition of $\widehat{\pi}(!^l!^n(\Phi' \otimes \Psi'))$ enable us to reason that $\widehat{\pi}(!!!^n(\Phi' \otimes \Psi')) \subseteq_{Qbit} \widehat{\pi}(!!^n(\Phi' \otimes \Psi'))$ holds.

It is fairly easy to see from the definition of function $\widehat{\pi}$ that these arguments also apply (slightly modified) to the other induction cases of Φ being of the form $!^n(\Phi' \oplus \Psi')$, $!^n(\Phi' \multimap \Psi')$ and $!^n(\forall X{<:}\Phi_X.\Psi')$.

$\Diamond$

At this point, we have successfully established properties (i') and (ii') and thus also properties (i) and (ii) for all proved types $\Theta \vdash \Phi$.

Finally, to conclude the proof of Lemma 4.35, it remains to show

($\star$) for each type term $\Phi \in \mathcal{T}_{type}$ it holds $\widehat{\pi}\big(\Phi[Qbit/X]\big) =_{Qbit} \widehat{\pi}(\Phi)[Qbit/X]$, where

- $\Upsilon =_{Qbit} \Upsilon'$ holds if and only if $\Upsilon \subseteq_{Qbit} \Upsilon'$ and $\Upsilon' \subseteq_{Qbit} \Upsilon$,
- $\widehat{\pi}(\Phi)[Qbit/X]$ denotes set $\big\{(p, S[Qbit/X], m) \;\big|\; (p, S, m) \in \widehat{\pi}(\Phi)\big\}$, and
- we assume (without loss of generality, since we identify α-equivalent type terms) that there is no type abstraction $(\forall X{<:}\dots)$ in Φ, i.e. X does not appear as bound type variable in Φ.

This immediately entails for any proved type $\Theta \vdash \Psi$ that if $X \in nftyv(\Psi)$, then there exists $(p, X, m) \in \widehat{\pi}(\Psi)$ with $m \geq 1$ (due to property (i)) and thus $(p, Qbit, m) \in \widehat{\pi}(\Psi)[Qbit/X]$, by definition of $\widehat{\pi}(\Psi)[Qbit/X]$. Consequently, there exists $(p, Qbit, m) \in \widehat{\pi}(\Psi[Qbit/X])$, with $m \geq 1$ (due to ($\star$)). But this means (by contradiction to property (ii)) that type-in-context $\Theta' \vdash \Psi[Qbit/X]$ is not derivable for any type context Θ'.

[35] The fact that neither $!!!^n Qbit$ nor $!!^n Qbit$ can be derived as proved types does not pose an obstacle to our argument.

This (almost) finishes the proof of Lemma 4.35. The final step of proving $(\star)$ requires one last induction on the definition of function $\widehat{\pi}$:

Base cases: Let Φ be of the form $!^n C$ for some $C \in \{\mathit{Top}, \mathit{Unit}, \mathit{Qbit}\}$. Then $(!^n C)[\mathit{Qbit}/X] = !^n C$ and $(\star)$ trivially holds, since no type variable X appears in C.

Let $\Phi = !^n X$ for some type variable X. Then $\widehat{\pi}\big((!^n X)[\mathit{Qbit}/X]\big) = \big\{(\varepsilon, \mathit{Qbit}, n)\big\} = \big\{(\varepsilon, X[\mathit{Qbit}/X], n)\big\} = \widehat{\pi}(!^n X)[\mathit{Qbit}/X]$. This means, $(\star)$ holds in this case.

Induction cases: Let Φ be of the form $!^n(\Phi' \otimes \Psi')$. By definition of substitution and function $\widehat{\pi}$ we have

$$\begin{aligned}\widehat{\pi}\big((!^n(\Phi' \otimes \Psi'))[\mathit{Qbit}/X]\big) &= \widehat{\pi}\big(!^n(\Phi'[\mathit{Qbit}/X] \otimes \Psi'[\mathit{Qbit}/X])\big)\\ &= \underbrace{\big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Phi'[\mathit{Qbit}/X])\big\}}_{=:\ \pi_5}\\ &\quad \cup \underbrace{\big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Psi'[\mathit{Qbit}/X])\big\}}_{=:\ \pi_6} .\end{aligned}$$

and

$$\begin{aligned}\widehat{\pi}\big(!^n(\Phi' \otimes \Psi')\big)[\mathit{Qbit}/X] &= \big\{(1p, S[\mathit{Qbit}/X], m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Phi')\big\}\\ &\quad \cup \big\{(2p, S[\mathit{Qbit}/X], m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Psi')\big\}\\ &= \underbrace{\big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Phi')[\mathit{Qbit}/X]\big\}}_{=:\ \pi_7}\\ &\quad \cup \underbrace{\big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Psi')[\mathit{Qbit}/X]\big\}}_{=:\ \pi_8} .\end{aligned}$$

Taking $!^n \Phi'[\mathit{Qbit}/X] = (!^n \Phi')[\mathit{Qbit}/X]$ and $!^n \Psi'[\mathit{Qbit}/X] = (!^n \Psi')[\mathit{Qbit}/X]$ into account, induction then yields

$$\widehat{\pi}\big(!^n \Phi'[\mathit{Qbit}/X]\big) = \widehat{\pi}\big((!^n \Phi')[\mathit{Qbit}/X]\big) =_{\mathit{Qbit}} \widehat{\pi}\big(!^n \Phi'\big)[\mathit{Qbit}/X]$$

and

$$\widehat{\pi}\big(!^n \Psi'[\mathit{Qbit}/X]\big) = \widehat{\pi}\big((!^n \Psi')[\mathit{Qbit}/X]\big) =_{\mathit{Qbit}} \widehat{\pi}\big(!^n \Psi'\big)[\mathit{Qbit}/X] .$$

And since π_5, π_7 and π_6, π_8 are constructed in analogous ways, respectively, we find $(\star)$ holds in this case. We may argue along similar lines for the cases of $\Phi = !^n(\Phi' \oplus \Psi')$ and $\Phi = !^n(\Phi' \multimap \Psi')$.

Consider the case $\Phi = !^n(\forall Y{<:}\Phi_X.\Psi')$. We then have

$$\begin{aligned}\widehat{\pi}\big(!^n(\forall Y{<:}\Phi_X.\Psi')\big)[\mathit{Qbit}/X] &= \big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^k \Phi_X)[\mathit{Qbit}/X]\big\}\\ &\quad \cup \big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Psi')[\mathit{Qbit}/X]\big\} .\end{aligned}$$

To continue, we (formally) need to distinguish two cases:

Let $Y \neq X$. Then

$$\begin{aligned}\widehat{\pi}\big((!^n(\forall Y{<:}\Phi_Y.\Psi'))[\mathit{Qbit}/X]\big) &= \big\{(1p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^k \Phi_Y[\mathit{Qbit}/X])\big\}\\ &\quad \cup \big\{(2p, S, m) \;\big|\; (p, S, m) \in \widehat{\pi}(!^n \Psi'[Y'/Y][\mathit{Qbit}/X])\big\} ,\end{aligned}$$

where Y' is a fresh free type variable neither appearing freely in Φ_Y nor in Ψ' and with $X \neq Y' \neq Y$.
Again, $!^k \Phi_Y[\mathit{Qbit}/X] = (!^k \Phi_Y)[\mathit{Qbit}/X]$ and induction help to obtain $\widehat{\pi}(!^k \Phi_Y[\mathit{Qbit}/X]) =_{\mathit{Qbit}} \widehat{\pi}(!^k \Phi_Y)[\mathit{Qbit}/X]$. Although the second half of the union looks more complicated, it is still accessible through the same arguments. Thus, we have $!^n \Psi'[Y'/Y][\mathit{Qbit}/X] = (!^n \Psi'[Y'/Y])[\mathit{Qbit}/X]$, which (with the help of induction) allows us to deduce

$$\widehat{\pi}\big(!^n \Psi'[Y'/Y][\mathit{Qbit}/X]\big) = \widehat{\pi}\big((!^n \Psi'[Y'/Y])[\mathit{Qbit}/X]\big) =_{\mathit{Qbit}} \widehat{\pi}\big(!^n \Psi'[Y'/Y]\big)[\mathit{Qbit}/X] .$$

Looking at the definitions of $\widehat{\pi}\big((!^n(\forall Y{<:}\Phi_Y.\Psi))[\mathit{Qbit}/X]\big)$ and $\widehat{\pi}\big(!^n(\forall Y{<:}\Phi_Y.\Psi)\big)[\mathit{Qbit}/X]$ in the light of what we just found out, this gives us all we need to conclude that $(\star)$ holds in this case, as well.

We do not have to consider the case of $Y = X$ due to our assumption that X is not bound by any type abstraction in Φ. But it is worthwhile to note that without this assumption we could (in general) only prove an inclusion $\widehat{\pi}(\Phi[Qbit/X]) \subseteq_{Qbit} \widehat{\pi}(\Phi)[Qbit/X]$ here, which would be strict for certain type terms Φ.

□

This lemma is the key for our subsequent considerations in this subsection. What we almost immediately get from it is that there is no type context Θ such that $\Theta \vdash !\Phi$ is derivable, if Φ is a strictly linear type term. To make this clear, consider again $\Phi \in \mathcal{T}_{sl}$ and type term $\Phi_{Qbit \mapsto Z}$ where type variable Z does not appear in Φ and is not bound by a type abstraction in Φ. Due to these assumptions and the definition of substitution of free type variables (Definition 4.4), we get $\Phi \equiv_\alpha \Phi_{Qbit \mapsto Z}[Qbit/Z]$, ($\equiv_\alpha$ meaning the two type terms are α-equivalent). By Lemma 4.34 we know that $Z \in \mathit{nftyv}(!\Phi_{Qbit \mapsto Z})$. If $\Theta \vdash !\Phi$ were derivable, then so were $Z{<:}\mathit{Top}, \Theta \vdash !\Phi_{Qbit \mapsto Z}$ by Lemma 4.36 (which is stated and proven below). However, Lemma 4.35 yields that there is no type context Θ' such that $\Theta' \vdash !\Phi_{Qbit \mapsto Z}[Qbit/Z]$ is a proved type, since $Z \in \mathit{nftyv}(!\Phi_{Qbit \mapsto Z})$. Hence, there cannot be a Θ such that $\Theta \vdash !\Phi$ is derivable, which leads to a contradiction.

Lemma 4.36. *If $\Theta \vdash \Phi$ is a proved type and $Z \in \mathcal{V}_{type} \setminus |\Theta|$ is a type variable which does not appear in Φ and is not bound by a type abstraction in Φ, then $Z{<:}\mathit{Top}, \Theta \vdash \Phi_{Qbit \mapsto Z}$ is a proved type, as well.*

Proof. We first show $Z{<:}\mathit{Top}, \Theta'$ is consistent for any consistent type context $\Theta' = Y_1{<:}\Phi_1, \ldots, Y_k{<:}\Phi_k$. Since Θ' is consistent, we have the following proved types:

$$\begin{aligned} \vdash \Phi_1 \ , \\ Y_1{<:}\Phi_1 \vdash \Phi_2 \ , \\ Y_1{<:}\Phi_1, Y_2{<:}\Phi_2 \vdash \Phi_3 \ , \\ \vdots \\ Y_1{<:}\Phi_1, \ldots, Y_{k-1}{<:}\Phi_{k-1} \vdash \Phi_k \ . \end{aligned}$$

Since $\vdash \mathit{Top}$ is derivable by rule (Top type) starting from the empty type context, we have consistency of $Z{<:}\mathit{Top}$. Hence, we may apply rule (type weakening) in a step-by-step fashion to obtain proved types

$$\begin{aligned} \vdash \mathit{Top} \ , \\ Z{<:}\mathit{Top} \vdash \Phi_1 \ , \\ Z{<:}\mathit{Top}, Y_1{<:}\Phi_1 \vdash \Phi_2 \ , \\ Z{<:}\mathit{Top}, Y_1{<:}\Phi_1, Y_2{<:}\Phi_2 \vdash \Phi_3 \ , \\ \vdots \\ Z{<:}\mathit{Top}, Y_1{<:}\Phi_1, \ldots, Y_{k-1}{<:}\Phi_{k-1} \vdash \Phi_k \ . \end{aligned}$$

Having settled this matter, we now proceed with an induction on the derivation of $\Theta \vdash \Phi$.

Base cases: Suppose $\Theta \vdash !^n \mathit{Top}$ has been derived by (Top type). Then $(!^n \mathit{Top})_{Qbit \mapsto Z} = !^n \mathit{Top}$ and we can derive $Z{<:}\mathit{Top}, \Theta \vdash !^n \mathit{Top}$ by (type weakening), since we know $Z{<:}\mathit{Top}, \Theta$ is consistent.
The same argument holds for the case of rule (Unit type).

Assume $\Theta \vdash \mathit{Qbit}$ has been derived by (Qbit type). Then we can perform the derivation

$$\frac{\vdash Z{<:}\mathit{Top}, \Theta}{Z{<:}\mathit{Top}, \Theta \vdash Z} \text{ (linear type variable)} \ .$$

Induction cases: Assume $\Theta \vdash \Phi$ is of the form $\Theta \vdash !^n(\Phi' \otimes \Psi')$ and has been derived by rule (product type). Then we get proved types $Z{<:}\mathit{Top}, \Theta \vdash !^n {\Phi'}_{Qbit \mapsto Z}$ and $Z{<:}\mathit{Top}, \Theta \vdash !^n {\Psi'}_{Qbit \mapsto Z}$ by induction. Using these, we can perform derivation

$$\frac{Z{<:}\mathit{Top}, \Theta \vdash !^n {\Phi'}_{Qbit \mapsto Z} \qquad Z{<:}\mathit{Top}, \Theta \vdash !^n {\Psi'}_{Qbit \mapsto Z}}{Z{<:}\mathit{Top}, \Theta \vdash \underbrace{!^n({\Phi'}_{Qbit \mapsto Z} \otimes {\Psi'}_{Qbit \mapsto Z})}_{= \Phi_{Qbit \mapsto Z}}} \text{ (product type)} \ .$$

Similar arguments apply to the cases of rules (function types) and (sum type).

Suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\Xi, \Xi' \vdash \Phi' <: !Top \qquad \Xi, X<:\Phi', \Xi' \vdash !^n \Psi'}{\Xi, \Xi' \vdash !^n(\forall X<:\Phi'.\Psi')} \text{ (nonlinear-polymorphic type)},$$

where we have $\Xi, \Xi' = \Theta$. From consistency of the type context in the second premise (due to Lemma 4.18), we get proved type $\Xi \vdash \Phi'$, which we may weaken to $\Xi, \Xi' \vdash \Phi'$, since Ξ, Ξ' is consistent (due to Lemma 4.19 applied to the first premise). Induction then yields proved types $Z<:Top, \Xi, \Xi' \vdash \Phi'_{Qbit \mapsto Z}$ and $Z<:Top, \Xi, X<:\Phi', \Xi' \vdash !^n \Psi'_{Qbit \mapsto Z}$. By Corollary 4.17 we know Φ' is of the form $\Phi' = !\Phi''$, and obviously we then also have $\Phi'_{Qbit \mapsto Z} = !\Phi''_{Qbit \mapsto Z}$, i.e. replacing *Qbit* by Z preserves leading exponentials. Hence, we may apply rule (type bound replacement) to obtain proved type $Z<:Top, \Xi, X<:\Phi'_{Qbit \mapsto Z}, \Xi' \vdash !^n \Psi'_{Qbit \mapsto Z}$ from $Z<:Top, \Xi, X<:\Phi', \Xi' \vdash !^n \Psi'_{Qbit \mapsto Z}$. But this also means we can apply rules (*Top* supertype) and (! right) to $Z<:Top, \Xi, \Xi' \vdash \Phi'_{Qbit \mapsto Z}$ in order to derive proved subtype $Z<:Top, \Xi, \Xi' \vdash \Phi'_{Qbit \mapsto Z} <: !Top$. Putting these facts together, we can perform derivation

$$\frac{\begin{array}{c} Z<:Top, \Xi, \Xi' \vdash \Phi'_{Qbit \mapsto Z} <: !Top \\ Z<:Top, \Xi, X<:\Phi'_{Qbit \mapsto Z}, \Xi' \vdash !^n \Psi'_{Qbit \mapsto Z} \end{array}}{Z<:Top, \Xi, \Xi' \vdash \underbrace{!^n(\forall X<:\Phi'_{Qbit \mapsto Z}.\Psi'_{Qbit \mapsto Z})}_{= \Phi_{Qbit \mapsto Z}}} \text{ (nonlinear-polymorphic type)}.$$

A similar, but somewhat easier argument yields the analogous result in case $\Theta \vdash \Phi$ being derived by (linear-polymorphic type).

□

By now we have made a good step forward in showing that our type system prohibits certain type terms to become parts of proved types. At this point, we have shown proved types $\Theta \vdash !^{n+1}\Phi$ cannot exist for type terms $\Phi \in \mathcal{T}_{sl}$ and $n \geq 0$. But we are not done yet, since so far we have not considered what happens if $!\Phi$ occurs as a subexpression in a type term Ψ, for instance. We actually do not have a formal notion of such subexpressions at the moment. Let us therefore formalize the intuitive concept of subterms of a type term.

Definition 4.37 (subterms of type terms).

We capture the notion of subterms of a type term by function $subt : \mathcal{T}_{type} \to \mathcal{P}(\mathcal{T}_{type})$ which we recursively define as follows
(for all $C \in \{Top, Unit, Qbit\}$, $X \in \mathcal{V}_{type}$, $\square \in \{\otimes, \oplus, \multimap\}$ and $\Phi, \Phi_X, \Psi \in \mathcal{T}_{type}$):

$$\begin{aligned} subt(C) &= \{C\} \\ subt(X) &= \{X\} \\ subt(!\Phi) &= subt(\Phi) \cup \{!\Phi\} \\ subt((\Phi \square \Psi)) &= subt(\Phi) \cup subt(\Psi) \cup \{(\Phi \square \Psi)\} \\ subt((\forall X<:\Phi_X.\Psi)) &= subt(\Phi_X) \cup subt(\Psi) \cup \{(\forall X<:\Phi_X.\Psi)\} \end{aligned}$$

There is not much to say about this definition besides that it captures the natural intuition.

More interesting is, however, an inspection of the derivation rules for proved types when we turn our attention to subterms of a proved type $\Theta \vdash \Phi$ (and by this we mean the subterms of type term Φ). Doing so, we notice that each of Φ's subterms has been itself part of a proved type that occurred during the derivation of $\Theta \vdash \Phi$. Consider for instance proved type $\Theta \vdash (\forall X<:\Phi_X.(\Psi_1 \otimes \Psi_2))$. During its derivation, we have derived proved types $\Xi, X<:\Phi_X, \Xi' \vdash \Psi_1$, $\Xi, X<:\Phi_X, \Xi' \vdash \Psi_2$ and $\Xi \vdash \Phi_X$ and so on, with $\Xi, \Xi' = \Theta$. Another example is proved type $\Theta \vdash !^n(\Phi' \otimes \Psi')$. When we derive it, we inevitably come across proved types $\Theta \vdash !^n\Phi'$ and $\Theta \vdash !^n\Psi'$, but we do in fact not derive $\Theta \vdash \Phi'$ and $\Theta \vdash \Psi'$ which are proved variants of subterms of $!^n(\Phi' \otimes \Psi')$. However, at this point rule (! elimination) supports our claim that we have *almost* derived subterms of the whole proved type.

Of course, these examples do not *prove* anything, but they give a good intuition for the following fact.

Fact 4.38. *If $\Theta \vdash \Phi$ is a proved type, then for each subterm $\Sigma \in subt(\Phi)$ there exists a type context Θ_Σ such that $\Theta_\Sigma \vdash \Sigma$ is a proved type.*

We will show the validity of this fact later on in Lemma 4.46(i). At this later point, we will even show a stronger result, but for now this fact is quite enough to show the final result of this section.

We conclude this section with the following result. It confirms that our type system indeed possesses a key feature that we aimed at from the beginning. This theorem thus constitutes one of the highlights of the present work.

Theorem 4.39 (enforceable linearity with respect to set $\mathcal{T}_{sl}$)**.**
Let $\Psi \in \mathcal{T}_{type}$ be a type term and let $\Phi \in \mathcal{T}_{sl}$ be a strictly linear type term. If we have $!^{n+1}\Phi \in subt(\Psi)$ with $n \geq 0$, then there is no type context Θ for which we can derive type-in-context $\Theta \vdash \Psi$ as a proved type.

Proof. Suppose $\Theta \vdash \Psi$ were derivable for an appropriate type context Θ. Then there exists a type context Θ_Φ according to Fact 4.38 such that $\Theta_\Phi \vdash !^{n+1}\Phi$ is a proved type.

Let Z be a type variable which does not appear in Φ and is not bound by a type abstraction in Φ. Consider type term $\Phi_{Qbit \mapsto Z}$ which we obtain from Φ by replacing all occurrences of type constant $Qbit$ by Z. As we have already argued before (right after the proof of Lemma 4.35), type terms Φ and $\Phi_{Qbit \mapsto Z}[Qbit/Z]$ are α-equivalent. Due to Lemma 4.34 we conclude $Z \in nftyv(!^{n+1}\Phi_{Qbit \mapsto Z})$. Since $\Theta_\Phi \vdash !^{n+1}\Phi$ is a proved type, we also have proved type $Z{<:}Top, \Theta_\Phi \vdash !^{n+1}\Phi_{Qbit \mapsto Z}$ due to Lemma 4.36. However, Lemma 4.35 yields that there is no type context Θ' such that $\Theta' \vdash !^{n+1}\Phi_{Qbit \mapsto Z}[Qbit/Z]$ is a proved type, since $Z \in nftyv(!^{n+1}\Phi_{Qbit \mapsto Z})$. Hence, there cannot be a Θ_Φ such that $\Theta_\Phi \vdash !^{n+1}\Phi$ is derivable. Obviously, we have a contradiction and thus $\Theta \vdash \Psi$ cannot be derived for any type context Θ. □

With this theorem we finish our investigation of strictly linear type terms in that we have shown our type system *cannot* derive duplicable versions of them.

It is a noteworthy consequence that the sort of types we dealt with in Theorem 4.39 is not the only sort of non-derivable types in our type system. What we also exclude is a preliminary stage of such types, namely the ones of the form $\Phi \in \{(\forall X{<:}\Phi_X.\Psi') \in \mathcal{T}_{type} \mid \Phi_X \not<: !Top \text{ and } X \in nftyv(\Psi')\}$. Consider, for instance, type term $(\forall X{<:}Top.!(X \otimes !X))$ which fits into this class of type terms and which does not have a strictly linear subterm. Now imagine a type application which results in a type term $!(X \otimes !X)[Qbit/X] = !(Qbit \otimes !Qbit)$. After such a type application, we end up with a type term that has strictly linear subterms *with* leading exponentials in front of it. For that reason, we also need to prohibit the above mentioned sorts of types to become proved types. We can convince ourselves of the underivability of these types by inspection of rules (linear-polymorphic type) and (nonlinear-polymorphic type). While the latter derivation rule only allows duplicable type bounds in type abstractions, the former derivation rule allows such type bounds but does not allow the bound type variable to occur nonlinearly in the scope. Hence, the above mentioned sort of types is clearly not derivable.

4.2.4 Transitivity of subtype derivations

When we have been discussing the derivation rules for proved subtypes, we already gave a hint that we can only achieve a weak form of transitivity for proved subtypes. More precisely, this means there does not exist a proved subtype $\Theta \vdash \Phi <: \Psi$ for *all* pairs of proved subtypes $\Theta \vdash \Phi <: \Upsilon$ and $\Theta \vdash \Upsilon <: \Psi$. One example has already been mentioned: we can derive $\vdash (!Qbit \otimes !Qbit) <: (Qbit \otimes Qbit)$ and $\vdash (Qbit \otimes Qbit) <: Top$ but not $\vdash (!Qbit \otimes !Qbit) <: Top$. The reason for this is rather simple: $\vdash (!Qbit \otimes !Qbit)$ is not a proved type, and thus we cannot apply rule (*Top* supertype) to it and the other derivation rules do not lead to the desired result. On the other hand, we may ask why (and how) can we derive a proved subtype $\Theta \vdash \Phi <: \Psi$ if we are given proved subtypes $\Theta \vdash \Phi <: \Upsilon$ and $\Theta \vdash \Upsilon <: \Psi$, and if $\Theta \vdash \Phi$ and $\Theta \vdash \Psi$ are proved types? The answer to this question is not so simple and easy to give. It turns out that we by now just know too little about the derivation of proved subtypes and what particular requirements the type contexts must fulfill from which we start this derivation. We will gain the necessary knowledge in the course of this subsection, however.

But before we dive deeper into derivations of proved types and proved subtypes, we take a look at minimal type contexts in the first part of this subsection. This will turn out to be a quite useful notion since it brings us into a position where we can concentrate on the actually necessary part of a type context.

Definition 4.40 (minimal type contexts).
Let $\Theta \vdash \Phi$ be a proved type. We call Θ *minimal* (with respect to $\Theta \vdash \Phi$), if $\Theta|_{|\Theta|\setminus V} \vdash \Phi$ cannot be derived for any nonempty set $V \subseteq |\Theta|$.
We analogously use this notion for type contexts Θ in proved subtypes $\Theta \vdash \Phi <: \Psi$.

Although we will not point it out explicitly in each case in the future, minimality of a type context is always meant with respect to a certain proved type or proved subtype (otherwise, a minimal type context would always be equal to the least (consistent) type context, i.e. the empty type context). If not stated explicitly, it will be clear from the respective context of discussion to which proved type or proved subtype a minimal type context is associated in this sense.

As the name suggests, we cannot reduce a minimal type context $\Theta_{\min}$ any further and still derive the associated proved type $\Theta_{\min} \vdash \Phi$. This immediately leads to the following chain of arguments: from Lemma 4.20 we know $\mathit{ftyv}(\Phi) \subseteq |\Theta|$ is a necessary condition for *any* type context Θ, in order to get proved type $\Theta \vdash \Phi$. Hence, for *minimal* type context $\Theta_{\min}$ there is no $\Theta' \sqsubseteq \Theta_{\min}$ that is well-scoped, shorter than $\Theta_{\min}$ and whose domain $|\Theta'|$ does contain all free type variables occurring in Φ (otherwise $\Theta_{\min}$ would not be minimal, due to item (i) in Corollary 4.28). Nevertheless, $\Theta_{\min}$ may still contain well-scoped (nontrivial) subsequences shorter than $\Theta_{\min}$ itself. Of course, the same holds for cases where $\Theta_{\min}$ is minimal with respect to a proved subtype.

This observation lays the ground for the following three quite technically flavored results.

Proposition 4.41. *Let Θ_Φ and Θ_Ψ be minimal type contexts with respect to proved types $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$, respectively, and let Φ and Ψ have the same free type variables, i.e. $\mathit{ftyv}(\Phi) = \mathit{ftyv}(\Psi)$. If there is a type context Θ such that $\Theta_\Phi \sqsubseteq \Theta$ and $\Theta_\Psi \sqsubseteq \Theta$, then $\Theta_\Phi = \Theta_\Psi$.*

Proof. The assumption $\Theta_\Phi \sqsubseteq \Theta \sqsupseteq \Theta_\Psi$ already gives us $\Theta_\Phi(Y_\Phi) = \Theta(Y_\Phi)$ for all $Y_\Phi \in |\Theta_\Phi|$ and $\Theta_\Psi(Y_\Psi) = \Theta(Y_\Psi)$ for all $Y_\Psi \in |\Theta_\Psi|$. Hence, we already have $\Theta_\Phi(Y) = \Theta_\Psi(Y)$ for all $Y \in |\Theta_\Phi| \cap |\Theta_\Psi|$, i.e. $\Theta_\Phi|_{|\Theta_\Phi|\cap|\Theta_\Psi|} = \Theta_\Psi|_{|\Theta_\Phi|\cap|\Theta_\Psi|}$.

It thus remains to show $|\Theta_\Phi| = |\Theta_\Psi|$. By Lemma 4.18 we know Θ_Φ and Θ_Ψ are consistent and thus also well-scoped, due to Corollary 4.21. By Lemma 4.20 we get $\mathit{ftyv}(\Phi) \subseteq |\Theta_\Phi|$ and $\mathit{ftyv}(\Psi) \subseteq |\Theta_\Psi|$, but then our assumption $\mathit{ftyv}(\Phi) = \mathit{ftyv}(\Psi)$ immediately yields $\mathit{ftyv}(\Phi) \subseteq |\Theta_\Phi| \cap |\Theta_\Psi|$. Since Θ_Φ and Θ_Ψ are well-scoped, we conclude that $\Theta_\Phi|_{|\Theta_\Phi|\cap|\Theta_\Psi|}$ and $\Theta_\Psi|_{|\Theta_\Phi|\cap|\Theta_\Psi|}$ are well-scoped, as well. But then, we can derive proved type $\Theta_\Phi|_{|\Theta_\Phi|\cap|\Theta_\Psi|} \vdash \Phi$ by Lemma 4.27, where $\Theta_\Phi|_{|\Theta_\Phi|\cap|\Theta_\Psi|} = \Theta_\Phi|_{|\Theta_\Phi|\setminus(|\Theta_\Phi|\setminus(|\Theta_\Phi|\cap|\Theta_\Psi|))}$. Hence, $(|\Theta_\Phi| \setminus (|\Theta_\Phi| \cap |\Theta_\Psi|))$ must be empty, since otherwise this would contradict minimality of Θ_Φ. Analogous reasoning applies to Θ_Ψ.

Consequently, we have $|\Theta_\Phi| = |\Theta_\Phi| \cap |\Theta_\Psi| = |\Theta_\Psi|$. □

Suppose we are given a proved type $\Theta \vdash \Phi$ and we intend to find a type context $\Theta_{\min}$ with $\Theta_{\min} \sqsubseteq \Theta$, so that we can derive $\Theta_{\min} \vdash \Phi$ and $\Theta_{\min}$ is minimal. Then the just proven proposition shows that $\Theta_{\min}$ is unique (modulo α-equivalence).

Obviously, we can easily extend Proposition 4.41 to the setting of proved subtypes.

Corollary 4.42. *Let Θ_1 and Θ_2 be minimal type contexts with respect to proved subtypes $\Theta_1 \vdash \Phi_1 <: \Psi_1$ and $\Theta_2 \vdash \Phi_2 <: \Psi_2$, respectively, and let $\mathit{ftyv}(\Phi_1) \cup \mathit{ftyv}(\Psi_1) = \mathit{ftyv}(\Phi_2) \cup \mathit{ftyv}(\Psi_2)$. If there is a type context Θ such that $\Theta_1 \sqsubseteq \Theta$ and $\Theta_2 \sqsubseteq \Theta$, then $\Theta_1 = \Theta_2$.*

Proof. The proof is analogous to the proof of Proposition 4.41, using Lemma 4.19 instead of Lemma 4.18, and Lemma 4.22 instead of Lemma 4.20. □

In the following proposition we meet a mixed situation compared to the two preceding results. What we learn from it is if we have minimal type contexts Θ_Φ, Θ_Ψ and Θ with respect to proved types $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$ and proved subtype $\Theta \vdash \Phi <: \Psi$, respectively, then we can conclude by their minimality that their domains are identical.

Proposition 4.43. *Let $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$ be proved types and $\Theta \vdash \Phi <: \Psi$ be a proved subtype, where Θ_Φ, Θ_Ψ and Θ are minimal. Then $|\Theta| = |\Theta_\Phi| \cup |\Theta_\Psi|$.*

Proof. From Lemmas 4.18 and 4.19 we get consistency of Θ_Φ, Θ_Ψ and Θ. Corollary 4.21 then further entails Θ_Φ, Θ_Ψ and Θ are well-scoped.

Moreover, Lemmas 4.20 and 4.22 yield the three results

- $\mathit{ftyv}(\Phi) \subseteq |\Theta_\Phi|$,
- $\mathit{ftyv}(\Psi) \subseteq |\Theta_\Psi|$ and
- $\mathit{ftyv}(\Phi) \cup \mathit{ftyv}(\Psi) \subseteq |\Theta|$.

According to item (i) in Corollary 4.28, minimality of Θ_Φ means there is no well-scoped (and consistent) subsequence $\Theta'_\Phi \sqsubseteq \Theta_\Phi$ shorter than Θ_Φ with $\mathit{ftyv}(\Phi) \subseteq |\Theta_\Phi|$. Of course, we have an analogous meaning for the minimality of Θ_Ψ and Θ.

In this sense, minimality and well-scopedness of Θ_Φ and $\mathit{ftyv}(\Phi) \subseteq |\Theta_\Phi|$ lead to subsets $S_0, \ldots, S_k \subseteq |\Theta_\Phi|$ and a sequence

$$S_0 \subseteq S_1 \subseteq \ldots \subseteq S_k \ ,$$

where $S_0 = \mathit{ftyv}(\Phi)$ and $S_i = S_{i-1} \cup \{\mathit{ftyv}(\Theta_\Phi(Y)) \mid Y \in S_{i-1}\}$ for all i, $1 \leq i \leq k$, and $k \geq 1$ is such that $S_{k-1} = S_k$ is a fixed point. Minimality of Θ_Φ then yields $S_k = |\Theta_\Phi|$, since otherwise we would have a well-scoped (and thus consistent) subsequence of Θ_Φ which contains $\mathit{ftyv}(\Phi)$. Obviously, an analogous chain can be build for Θ_Ψ.

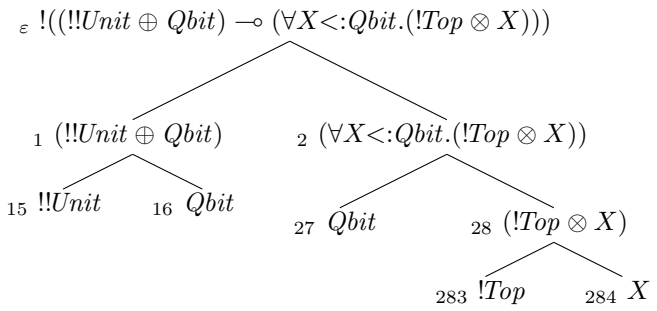
But due to $\mathit{ftyv}(\Phi) \cup \mathit{ftyv}(\Psi) \subseteq |\Theta|$ and due to Θ's well-scopedness, we can find exactly the same chains as for Θ_Φ and Θ_Ψ also in the domain of Θ. And since Θ is also assumed to be minimal, we thus have $|\Theta_\Phi| \cup |\Theta_\Psi| = |\Theta|$. □

Having gained these three basic results, we know enough about minimal type contexts for the moment.

Let us next introduce the key tool for the remainder of this section. The underlying idea goes back to the representation of syntactic expressions such as type terms by trees. Usually, each subexpression of the full syntactic expression is then represented by a node in the tree, while the root represents the full expression. We will, however, modify this idea according to our needs. Firstly, we do not need an explicit representation of those trees (the idea of a tree is rather useful for our intuition but nothing more). What we really need instead are addresses that we (in principle) assign to the nodes in such a tree. Secondly, we use a coarser grained division of type terms into subexpressions than function *subt* offers. To illustrate this, consider a type term $!^n\Phi$, where Φ shall be linear. According to set $\mathit{subt}(\Phi)$ of subterms of Φ, we would have one tree node for each $!^i\Phi$ with $0 \leq i \leq n$. What we need instead is one node corresponding to the full $!^n\Phi$ and then continue to build nodes corresponding to the subterms of Φ (unequal to Φ itself).

Here is an example of what we aim at:

Consider type term $!((!!\mathit{Unit} \oplus \mathit{Qbit}) \multimap (\forall X{<:}\mathit{Qbit}.(!\mathit{Top} \otimes X)))$. In the corresponding coarse syntax tree the addresses of subexpressions are written as subscripts to the left of each node:



As we have already emphasized, the tree representation itself is not really of importance, but rather supports our intuition. What indeed is important, however, is the addressing of all the involved subexpressions. For addresses we use finite words over alphabet $\{1, \ldots, 8\}$, and each sort of composite type term is assigned a pair of digits:

- function type ($\Phi \multimap \Psi$): 1 for Φ, 2 for Ψ;
- product type ($\Phi \otimes \Psi$): 3 for Φ, 4 for Ψ;
- sum type ($\Phi \oplus \Psi$): 5 for Φ, 6 for Ψ;
- type abstraction ($\forall X{<:}\Phi_X.\Psi$): 7 for Φ_X, 8 for Ψ.

The following definition of function *sta* (subterm addressing) puts the just explained intuitions into a formal frame.

Definition 4.44 (subterm addressing – function *sta*).
We recursively define function $sta : \mathcal{T}_{type} \to \mathcal{P}\big(\{1,\ldots,8\}^* \times \mathcal{T}_{type}\big)$ as follows
(for all $C \in \{Top, Unit, Qbit\}$, $X \in \mathcal{V}_{type}$ and $\Phi, \Phi_X, \Psi \in \mathcal{T}_{type}$)

$$\begin{aligned}
sta\big(C\big) &:= \big\{(\varepsilon, C)\big\}\ ,\\
sta\big(X\big) &:= \big\{(\varepsilon, X)\big\}\ ,\\
sta\big(!\Phi\big) &:= \big\{(p,\Phi') \in sta(\Phi_0) \setminus \{(\varepsilon,\Phi_0)\} \;\big|\; \text{where } \Phi = !^n\Phi_0 \text{ for linear } \Phi_0\big\}\\
&\quad \cup \big\{(\varepsilon, !\Phi)\big\}\ ,\\
sta\big((\Phi \multimap \Psi)\big) &:= \big\{(1p,\Phi') \;\big|\; (p,\Phi') \in sta(\Phi)\big\} \cup \big\{(2p,\Psi') \;\big|\; (p,\Psi') \in sta(\Psi)\big\}\\
&\quad \cup \big\{\big(\varepsilon, (\Phi \multimap \Psi)\big)\big\}\ ,\\
sta\big((\Phi \otimes \Psi)\big) &:= \big\{(3p,\Phi') \;\big|\; (p,\Phi') \in sta(\Phi)\big\} \cup \big\{(4p,\Psi') \;\big|\; (p,\Psi') \in sta(\Psi)\big\}\\
&\quad \cup \big\{\big(\varepsilon, (\Phi \otimes \Psi)\big)\big\}\ ,\\
sta\big((\Phi \oplus \Psi)\big) &:= \big\{(5p,\Phi') \;\big|\; (p,\Phi') \in sta(\Phi)\big\} \cup \big\{(6p,\Psi') \;\big|\; (p,\Psi') \in sta(\Psi)\big\}\\
&\quad \cup \big\{\big(\varepsilon, (\Phi \oplus \Psi)\big)\big\}\ ,\\
sta\big((\forall X{<:}\Phi_X.\Psi)\big) &:= \big\{(7p,\Phi') \;\big|\; (p,\Phi') \in sta(\Phi)\big\} \cup \big\{(8p,\Psi') \;\big|\; (p,\Psi') \in sta(\Psi)\big\}\\
&\quad \cup \big\{\big(\varepsilon, (\forall X{<:}\Phi_X.\Psi)\big)\big\}\ .
\end{aligned}$$

At first, we claim (and prove) the division into subexpression done by function *sta* produces subterms.

Proposition 4.45. *Given a type term Φ, it holds $\big\{\Sigma \mid (p,\Sigma) \in sta(\Phi)\big\} \subseteq subt(\Phi)$.*

Proof. Let us abbreviate $\big\{\Sigma \mid (p,\Sigma) \in sta(\Phi)\big\}$ by $\pi_r(sta(\Phi))$ (referring to the idea of projections in the right component of pairs).
Induction on the structure of Φ:

Base cases: Let $\Phi \in \{Top, Unit, Qbit\} \cup \mathcal{V}_{type}$. Then $sta(\Phi) = \big\{(\varepsilon,\Phi)\big\}$ and $subt(\Phi) = \{\Phi\}$. Hence, our claim holds in this case.

Induction cases: Let $\Phi = !^n\Phi_0$ for linear Φ_0. Then

$$sta(!^n\Phi_0) = \underbrace{\big\{(p,\Phi') \in sta(\Phi_0) \setminus \{(\varepsilon,\Phi_0)\}\big\}}_{\subset\, sta(\Phi_0)} \cup \big\{(\varepsilon, !^n\Phi_0)\big\}$$

and $subt(!^n\Phi_0) = subt(\Phi_0) \cup \big\{!^i\Phi_0 \;\big|\; 0 \le i \le n\big\}$. On the one hand, we get $\pi_r(sta(\Phi_0)) \subseteq subt(\Phi_0)$ by induction. And since $!^n\Phi_0 \in subt(\Phi)$ holds on the other hand, we then know $\pi_r(sta(!^n\Phi_0)) \subseteq subt(!^n\Phi_0)$.

Let $\Phi = (\Phi' \square \Psi')$ for any $\square \in \{\multimap, \otimes, \oplus\}$. Then

$$sta((\Phi'\square\Psi')) = \underbrace{\big\{(kp,\Phi'') \;\big|\; (p,\Phi'') \in sta(\Phi')\big\}}_{=\, s_1} \cup \underbrace{\big\{(k'p,\Psi'') \;\big|\; (p,\Psi'') \in sta(\Psi')\big\}}_{=\, s_2} \cup \big\{(\varepsilon,(\Phi'\square\Psi'))\big\}$$

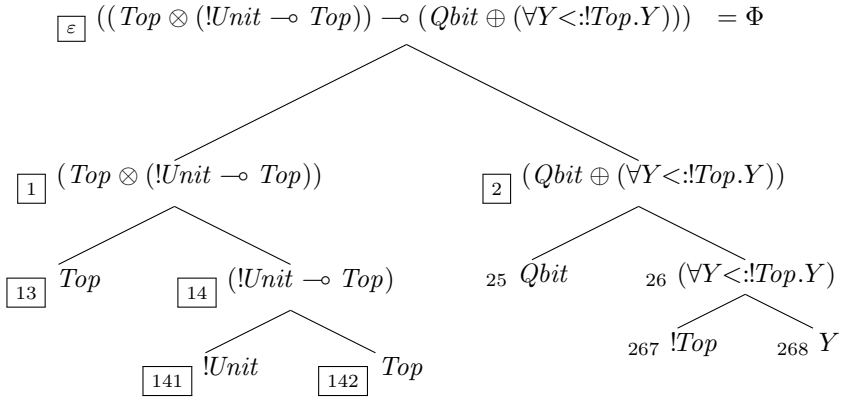
and $subt((\Phi'\square\Psi')) = subt(\Phi') \cup subt(\Psi') \cup \{(\Phi'\square\Psi')\}$.
By induction, we then get $\pi_r(s_1) = \pi_r(sta(\Phi')) \subseteq subt(\Phi')$ and $\pi_r(s_2) = \pi_r(sta(\Psi')) \subseteq subt(\Psi')$, since s_1 and s_2 take over all pairs from $sta(\Phi')$ and $sta(\Psi')$, respectively, and just modify the first components of these pairs. Hence, we have $\pi_r(sta((\Phi'\square\Psi'))) \subseteq subt((\Phi'\square\Psi'))$.

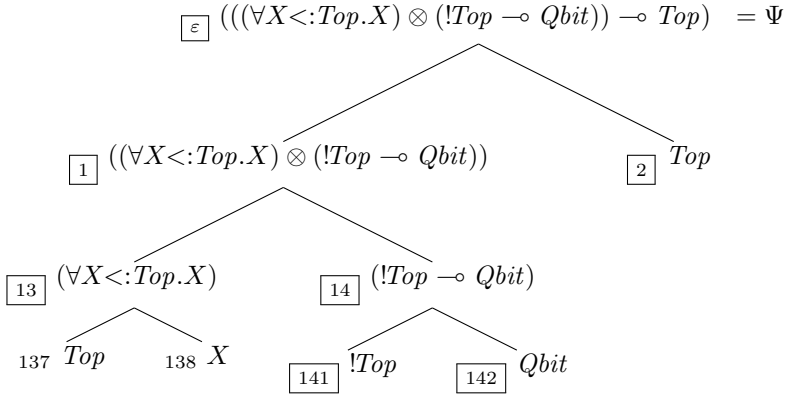
Case $\Phi = (\forall X{<:}\Phi_X.\Psi')$ is handled analogously. □

Let us next investigate an example to see what subterm addressing as it is done by function *sta* tells about subtype derivations. Consider the following subtype derivation of $\vdash \Phi <: \Psi$

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash (\forall X{<:}Top.X)}}{\vdash (\forall X{<:}Top.X) <: Top} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash !Unit}}{\vdash !Unit <: Top}}{\vdash !Unit <: !Top} \qquad \dfrac{\dfrac{\vdash \emptyset}{\vdash Qbit}}{\vdash Qbit <: Top}}{\vdash (!Top \multimap Qbit) <: (!Unit \multimap Top)}}{\vdash ((\forall X{<:}Top.X) \otimes (!Top \multimap Qbit)) <: (Top \otimes (!Unit \multimap Top))} \qquad \dfrac{\dfrac{\vdots}{\vdash (Qbit \oplus (\forall Y{<:}!Top.Y))}}{\vdash (Qbit \oplus (\forall Y{<:}!Top.Y)) <: Top}}{\vdash ((Top \otimes (!Unit \multimap Top)) \multimap (Qbit \oplus (\forall Y{<:}!Top.Y))) <: (((\forall X{<:}Top.X) \otimes (!Top \multimap Qbit)) \multimap Top)}$$

where we omitted labels for derivation rules to save space. We can easily find derivations for proved types $\vdash (\forall X{<:}Top.X)$ and $\vdash (Qbit \oplus (\forall Y{<:}!Top.Y))$ but do not give them explicitly here. Although such derivation trees are the proper way to derive proved subtypes in a faithful way (checking validity of premises and drawing valid conclusions according to the derivation rules given in Definition 4.15), they do not give a clear view of all interesting aspects of subtype derivations. To clear our view up a bit, we take a look at the subterm addressing of the two type terms involved, where we put frames around subterm addresses that are common to both type terms Φ and Ψ.

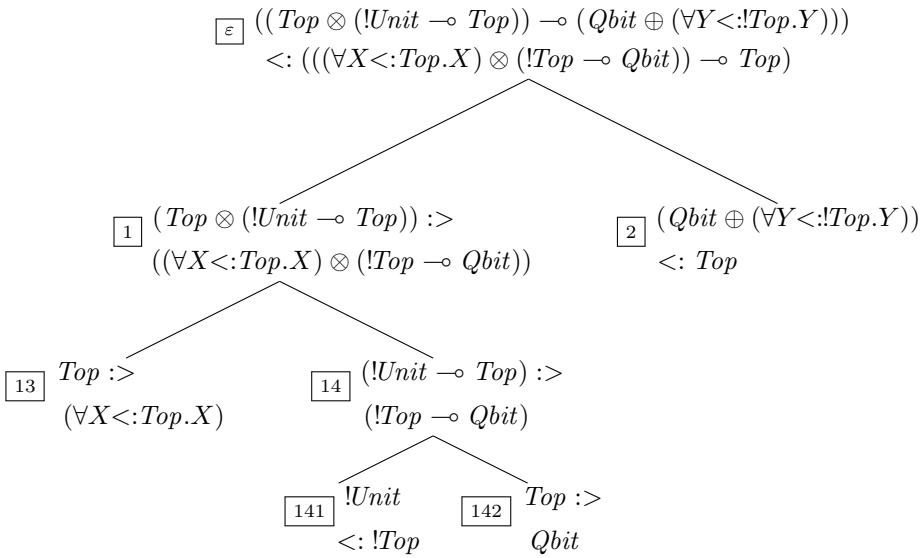




From this we notice that each of the two type terms in a subtyping statement $\Phi <: \Psi$ can show substructures which do not appear in the other. In the example addresses 13 and 2 are such points. On the one hand, we find type term *Top* at address 13 in Φ, and in Ψ we find $(\forall X{<:}Top.X)$ at this address. On the other hand, Ψ has subterm *Top* at address 2, where Φ shows the more complex $(Qbit \oplus (\forall Y{<:}!Top.Y))$. Finding *Top* at such common addresses showing different term structures is by no means a coincidence. In fact, this is a consequence of the derivation rules for proved subtypes. As we can see by inspection of these rules, only rule (*Top* supertype) allows the introduction of more complex structures in the left-hand side type term than in the right-hand side one (left and right of $<:$).[36] This immediately explains the

[36] We will use this knowledge in section 4.3 to restrict certain subtype derivations in order to guarantee the same structure on both sides.

"hidden" (or "lost", depending on the point of view) structure at address 2 in the example. The situation at address 13 already gives a hint towards another aspect which is of great importance for our subsequent considerations. Since there the direction of hiding structure is reversed in the sense that Ψ shows a more complex structure at address 13 than Φ does, although Φ is subtype and not supertype of Ψ. The reason for this is the principal $\multimap$ symbol in Φ and Ψ, because the first premise of rule (function type) shows this reversed direction of subtyping, in contrast to the second premise of the same rule, for instance. (We have motivated this reversed direction in the discussion of the axioms of our subtype relation right after Definition 4.7.) To illustrate this, we lay the two subterm addressing trees one over another and explicitly mark the direction of subtyping. In each node the upper subterm stems from Φ and the lower subterm is taken from Ψ, respectively.



What we discover here is of great importance for the key result in the current subsection, namely Lemma 4.46. The subtyping statement at position ε is the overall statement which we have, namely the one in proved subtype $\vdash \Phi <: \Psi$. At addresses 2 and 141, we have the same direction of subtyping, namely a subterm of Φ is a subtype of a subterm of Ψ. Addresses 1, 13, 14 and 142, however, show the reversed direction of subtyping. We have already pointed out that this is because of the application of derivation rule (function type) where the first premise causes this reversal of direction. This reversal is reflected by the single digit 1 in all these addresses. But what happened at position 141? Well, there the direction of subtyping has been reversed twice, as we can easily check by looking at the number of digits 1 in the address. Hence – and this is the essence of the current discussion – we can directly read from the respective addresses, how often the direction of subtyping has been reversed during the derivation of the proved subtype. That means when we count the digits 1 and 7 (recall that rule (polymorphic subtype) also exhibits a reversal of direction of subtyping in its first premise) in an address p common to Φ and Ψ and end up with an odd number, then the subterm of Φ at address p is a *super*type of the subterm of Ψ at address p, and a *sub*type in case of an even number of digits 1 and 7. This is the key observation explaining the form of items (ii.2), (ii.3), (iv.2.2) and (iv.2.3) in Lemma 4.46. (What we just claimed will be proven formally in proposition (ii) in the mentioned lemma.)

In items (iv.1.1) and (iv.1.2) of the very same lemma we even go one step farther. But to explain this step, we first need another example – namely one where type contexts play a more important role. Consider the following type derivations of proved types $\Phi := (\forall X{<:}(\forall Y{<:}!\mathit{Unit}.\mathit{Top}).!\mathit{Unit})$ and $\Psi := (\forall X{<:}(\forall Y{<:}!\mathit{Top}.\mathit{Qbit}).!\mathit{Top})$ where we omit rule labels to the right of the horizontal lines, but write addresses to the left instead to indicate where the currently derived subterm finds its place in the full type term at the end. Some of these addresses are framed again, but we explain the special meaning of this below. Furthermore, we have (as an exception to common practice in the rest of our work) included full derivations of the consistency of type contexts and marked the implication of consistency by "⇓".

$$\dfrac{\boxed{\varepsilon}\ \dfrac{\vdash \emptyset \qquad {}^{8}\dfrac{\dfrac{\vdash \emptyset}{\vdash !Unit}\,{}^{77}}{\Downarrow}}{X{<:}(\forall Y{<:}!Unit.Top) \vdash !Unit}}{\underbrace{\vdash (\forall X{<:}(\forall Y{<:}!Unit.Top).!Unit)}_{=\Phi}}\ X \notin nftyv(!Unit)}{}$$

$$\dfrac{\boxed{7}\ \dfrac{\vdash \emptyset \qquad {}^{78}\dfrac{\dfrac{\vdash Y{<:}!Unit}{\Downarrow}}{Y{<:}!Unit \vdash Top}}{\vdash (\forall Y{<:}!Unit.Top)}\ Y \notin nftyv(Top)}{}$$

$$\dfrac{\boxed{\varepsilon}\ \dfrac{\vdash \emptyset \qquad {}^{8}\dfrac{\dfrac{\vdash \emptyset}{\vdash !Top}\,{}^{77}}{\Downarrow}}{X{<:}(\forall Y{<:}!Top.Qbit) \vdash !Top}}{\underbrace{\vdash (\forall X{<:}(\forall Y{<:}!Top.Qbit).!Top)}_{=\Psi}}\ X \notin nftyv(!Top)}{}$$

$$\dfrac{\boxed{7}\ \dfrac{\vdash \emptyset \qquad {}^{78}\dfrac{\dfrac{\vdash Y{<:}!Top}{\Downarrow}}{Y{<:}!Top \vdash Qbit}}{\vdash (\forall Y{<:}!Top.Qbit)}\ Y \notin nftyv(Qbit)}{}$$

Before we next look at a subtype derivation that brings the two just derived type terms together in proved subtype $\vdash \Phi <: \Psi$, we shall point out the reason for which two particular addresses have been framed. What we first realize is both framed addresses 7 and ε are associated to derivations of type abstractions (by rule (linear-polymorphic type) in both cases, but it could have been its nonlinear counterpart as well). The second obvious but still remarkable circumstance is which type context appears in the second premise of each of these rule instances. The crucial point is that in the derivation steps marked with address 7 we use type bound $!Unit$ for Y in the Φ case and type bound $!Top$ for Y in the Ψ case. Similarly, at root address ε we have type bound $(\forall Y{<:}!Unit.Top)$ assigned to X in the derivation of Φ and $(\forall Y{<:}!Top.Qbit)$ in the derivation of Ψ. Thinking about a derivation for proved subtype $\vdash \Phi <: \Psi$, we already know both type terms show subterms $(\forall Y{<:}\dots)$ and $(\forall X{<:}\dots)$ in corresponding addresses, respectively. Hence, the structure of these subterms cannot be "hidden behind Top". In other words, we have to use rule (polymorphic subtype) for the derivation of proved subtype $\vdash (\forall Y{<:}\dots) <: (\forall Y{<:}\dots)$ for address 7 and proved subtype $\vdash (\forall X{<:}\dots) <: (\forall X{<:}\dots)$ for address ε (see also the subtype derivation shown below). The problem that arises is the one we have just seen: during derivation of Φ and Ψ we assign different type bounds to type variable X in the respectively used type contexts, and the same is true for Y. However, the type context used for subtype derivation can only assign one of the two type bounds to X (and also to Y). Hence, we have to make a proper choice on how to build up the type context we want to use for subtype derivation. The key to the solution of this "type context puzzle" is again counting digits 1 and 7 in addresses (and checking whether their number is even or odd) and an inspection of rule (polymorphic subtype). To make this task a bit easier, we use our example and the following derivation of $\vdash \Phi <: \Psi$.

$$\dfrac{\boxed{\varepsilon}\ \dfrac{\boxed{7}\ \dfrac{{}^{77}\dfrac{\vdots}{\vdash !Unit <: !Top} \qquad {}^{78}\dfrac{\vdots}{Y{<:}!Unit \vdash Qbit <: Top}}{\vdash (\forall Y{<:}!Top.Qbit) <: (\forall Y{<:}!Unit.Top)} \qquad {}^{8}\dfrac{\vdots}{X{<:}(\forall Y{<:}!Top.Qbit) \vdash !Unit <: !Top}}{\vdash (\forall X{<:}(\forall Y{<:}!Unit.Top).!Unit) <: (\forall X{<:}(\forall Y{<:}!Top.Qbit).!Top)}}{}$$

Looking at this example, we notice the involved type contexts needed to assign type bound $!Unit$ to Y, which has also been used in the above derivation of Φ, and type bound $(\forall Y{<:}!Top.Qbit)$ to X, which has been used in the above derivation of Ψ. As already mentioned, both choices are connected with the respective addresses and the number of occurrences of digits 1 and 7 therein. Thus, if the address of a

type abstraction contains an even number of 1s and 7s, we use the type bound associated to Ψ, and in the odd case we take over the type bound associated to Φ. And this is exactly what is formulated formally in (iv.1.1) and (iv.1.2) in Lemma 4.46.

All the above described ideas culminate in the following lemma. It consists of four separate propositions which mark different stages which we have to pass consecutively, since they are partly based on one another. In this sense (iv) constitutes the climax of the lemma. So, for instance, (i) provides the pieces of the above mentioned "type context puzzle", while (ii) and (iii) together allow for drawing conclusions on the form of subterms of Ψ (or Φ), if we are given a subterm of Φ (or Ψ), its address and proved subtypes $\Theta \vdash \Phi <: \Upsilon$ and $\Theta \vdash \Upsilon <: \Psi$. In (iv) we eventually come to the point, where we solve the "type context puzzle", and show how the thus obtained type context can be used to construct subtype derivations for subterms of Φ and Ψ that share common addresses. And since Φ and Ψ are subterms of themselves, we eventually end up with a transitivity result. Along this road, we use minimal type contexts as often as necessary, since we can usually draw more conclusions from minimal type contexts than from arbitrary ones.

Lemma 4.46. *In each of the following propositions we assume none of the $Y \in |\Theta|$ appears bound in Φ or Ψ. Furthermore, we assume different type abstractions in Φ bind different type variables, and the same shall hold for Ψ.*[37]

(i) If $\Theta \vdash \Phi$ is a proved type with Θ being minimal, then for each subterm $\Sigma \in subt(\Phi)$ there exists a type context Θ_Σ such that

(i.1) $\Theta \sqsubseteq \Theta_\Sigma$ holds,

(i.2) $\Theta_\Sigma \vdash \Sigma$ is a proved type, and

(i.3) all $Y \in |\Theta_\Sigma| \setminus |\Theta|$ appear bound in Φ with assigned type bound $\Theta_\Sigma(Y)$, i.e. $(\forall Y{<:}\Theta_\Sigma(Y).\ \ldots) \in subt(\Phi)$.

(ii) If $\Theta \vdash \Phi <: \Psi$ is a proved subtype, where Θ is minimal, then for each pair $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in sta(\Phi) \times sta(\Psi)$ there exists a type context $\widehat{\Theta}$ such that

(ii.1) $\Theta \sqsubseteq \widehat{\Theta}$,

(ii.2) $\widehat{\Theta} \vdash \Sigma_\Phi <: \Sigma_\Psi$ is a proved subtype, if $|p|_1 + |p|_7 \mod 2 = 0$, and

(ii.3) $\widehat{\Theta} \vdash \Sigma_\Psi <: \Sigma_\Phi$ is a proved subtype, if $|p|_1 + |p|_7 \mod 2 = 1$.

(iii) Consider proved subtypes $\Theta_{\Phi\Upsilon} \vdash \Phi <: \Upsilon$ and $\Theta_{\Upsilon\Psi} \vdash \Upsilon <: \Psi$ and the sets $addr_\Phi := \big\{p \;\big|\; (p, \Sigma_\Phi) \in sta(\Phi)\big\}$ and $addr_\Psi$, $addr_\Upsilon$ (which shall be defined analogously to $addr_\Phi$). It then holds $addr_\Phi \cap addr_\Psi \subseteq addr_\Upsilon$, i.e. if there is a pair $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in sta(\Phi) \times sta(\Psi)$, then we have $(p, \Sigma_\Upsilon) \in sta(\Upsilon)$ for some subterm Σ_Υ of Υ.

(iv) Let $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$ be proved types with minimal type contexts Θ_Φ and Θ_Ψ, and Θ_Φ and Θ_Ψ be such that $\Theta_\Phi \sqsubseteq \Theta$ and $\Theta_\Psi \sqsubseteq \Theta$ hold for a common consistent type context Θ, which we assume to be as small as possible, i.e. $|\Theta| = |\Theta_\Phi| \cup |\Theta_\Psi|$. Moreover, let $\Theta_{\Phi\Upsilon} \vdash \Phi <: \Upsilon$ and $\Theta_{\Upsilon\Psi} \vdash \Upsilon <: \Psi$ be proved subtypes. Then there exists a consistent type context $\widehat{\Theta}_*$ with $\Theta \sqsubseteq \widehat{\Theta}_*$, such that all type variables $Y \in |\widehat{\Theta}_*| \setminus |\Theta|$ appear bound in Φ or Ψ with assigned type bound $\widehat{\Theta}_*(Y)$ and such that for each pair
$\big((p, (\forall X{<:}\Sigma_{\Phi_1}.\Sigma_{\Phi_2})), (p, (\forall X{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2}))\big) \in sta(\Phi) \times sta(\Psi)$ we have

(iv.1.1) $\widehat{\Theta}_*(X) = \Sigma_{\Psi_1}$ if $|p|_1 + |p|_7 \mod 2 = 0$ and

(iv.1.2) $\widehat{\Theta}_*(X) = \Sigma_{\Phi_1}$ if $|p|_1 + |p|_7 \mod 2 = 1$;

and for each $\big((p', \Sigma_\Phi), (p', \Sigma_\Psi)\big) \in sta(\Phi) \times sta(\Psi)$, there exists a smallest type context $\widehat{\Theta}$ such that

(iv.2.1) $\Theta \sqsubseteq \widehat{\Theta} \sqsubseteq \widehat{\Theta}_*$,

(iv.2.2) $\widehat{\Theta} \vdash \Sigma_\Phi <: \Sigma_\Psi$ is a proved subtype, if $|p'|_1 + |p'|_7 \mod 2 = 0$, and

[37] These are not very restrictive assumptions, since we identify α-equivalent type terms.

(iv.2.3) $\widehat{\Theta} \vdash \Sigma_\Psi <: \Sigma_\Phi$ *is a proved subtype, if* $|p'|_1 + |p'|_7 \mod 2 = 1$.

Proof. We first show (i) by induction on the derivation of $\Theta \vdash \Phi$.

Base cases: Suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\vdash \Theta}{\Theta \vdash !^n \mathit{Top}} \,(\mathit{Top}\text{ type})\ ,$$

where $\Phi = !^n \mathit{Top}$ for some $n \geq 0$. Then we have $\mathit{subt}(!^n \mathit{Top}) = \{!^i \mathit{Top} \mid 0 \leq i \leq n\}$. Since $\vdash \Theta$ is already given, we can use the same rule to derive all proved types $\Theta \vdash \Sigma$ with $\Sigma \in \mathit{subt}(!^n \mathit{Top})$. (Please note (i.1) and (i.3) trivially hold due to $\Theta \sqsubseteq \Theta$ and since $|\Theta| \setminus |\Theta| = \emptyset$.)
The same holds for rules (*Unit* type), (*Qbit* type), (linear type variable) and (nonlinear type variable).

Induction cases: Suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\Theta \vdash \Phi' \qquad \Theta \vdash \Psi'}{\Theta \vdash !^n(\Phi' \multimap \Psi')} \,(\text{function type})\ .$$

This entails $\mathit{subt}(!^n(\Phi' \multimap \Psi')) = \{!^i(\Phi' \multimap \Psi') \mid 0 \leq i \leq n\} \cup \mathit{subt}(\Phi') \cup \mathit{subt}(\Psi')$. Then we can derive all proved types $\Theta \vdash \Sigma$ with $\Sigma \in \{!^i(\Phi' \multimap \Psi') \mid 0 \leq i \leq n\}$ by (! elimination) applied to $\Theta \vdash !^n(\Phi' \multimap \Psi')$, or doing nothing at all in case of $n = 0$. That means, we use $\Theta_\Sigma = \Theta$, and thus (i.1) and (i.3) again trivially hold.
By induction, we moreover get for all $\Sigma_1 \in \mathit{subt}(\Phi')$ there exist type contexts Θ_{Σ_1} with $\Theta \sqsubseteq \Theta_{\Sigma_1}$, such that $\Theta_{\Sigma_1} \vdash \Sigma_1$ can be derived, and analogously for all $\Sigma_2 \in \mathit{subt}(\Psi')$. Hence, we are done in this case.
Similar reasoning applies to the cases of rule (product type) and (sum type).

Suppose $\Theta \vdash \Phi$ has been derived by

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi')} \,(\text{linear-polymorphic type})$$

with $\Xi, \Xi' = \Theta$. Then $\mathit{subt}((\forall X{<:}\Phi_X.\Psi')) = \mathit{subt}(\Phi_X) \cup \mathit{subt}(\Psi') \cup \{(\forall X{<:}\Phi_X.\Psi')\}$. By induction, we immediately get for each $\Sigma_2 \in \mathit{subt}(\Psi')$ there exists a type context Θ_{Σ_2} with $\Xi, \Xi' \sqsubseteq \Xi, X{<:}\Phi_X, \Xi' \sqsubseteq \Theta_{\Sigma_2}$, such that $\Theta_{\Sigma_2} \vdash \Sigma_2$ is a proved type, and such that all type variables $Y \in |\Theta_{\Sigma_2}| \setminus |\Xi, X{<:}\Phi_X, \Xi'|$ appear bound in Ψ'. But since type variable X is bound in $\Phi = (\forall X{<:}\Phi_X.\Psi')$ and type bound $\Theta_{\Sigma_2}(X) = \Phi_X$ is assigned to it in Θ_{Σ_2}, we thus know all $Y \in |\Theta_{\Sigma_2}| \setminus |\Xi, \Xi'|$ appear bound in Φ with assigned type bound $\Theta_{\Sigma_2}(Y)$.
From the second premise we get consistency of $\Xi, X{<:}\Phi_X, \Xi'$ by Lemma 4.18, leading us to proved type $\Xi \vdash \Phi_X$. Since we already have $\vdash \Xi, \Xi'$ from the first premise, we may apply (type weakening) to obtain proved type $\Xi, \Xi' \vdash \Phi_X$. Now induction gives us the desired type contexts Θ_{Σ_1} for each $\Sigma_1 \in \mathit{subt}(\Phi_X)$ with $\Xi, \Xi' \sqsubseteq \Theta_{\Sigma_1}$ and proved type $\Theta_{\Sigma_1} \vdash \Sigma_1$. Moreover, (i.3) holds due to this induction. Hence, we have covered the whole set $\mathit{subt}((\forall X{<:}\Phi_X.\Psi'))$, since proved type $\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi')$ is already given by assumption.

In case of (nonlinear-polymorphic type) we can cover the $\mathit{subt}(\Phi_X) \cup \mathit{subt}(\Psi')$ part of

$$\mathit{subt}(!^n(\forall X{<:}\Phi_X.\Psi')) = \left\{!^i(\forall X{<:}\Phi_X.\Psi') \mid 0 \leq i \leq n\right\} \cup \mathit{subt}(\Phi_X) \cup \mathit{subt}(\Psi')$$

analogously. However, here we get consistency of Ξ, Ξ' indirectly from the first premise by application of Lemma 4.19.
For the remaining part, we get proved types $\Xi, \Xi' \vdash !^i(\forall X{<:}\Phi_X.\Psi')$ for all i with $0 \leq i \leq n$ by rule (! elimination). And again, (i.1) and (i.3) trivially hold because of $\Xi, \Xi' \sqsubseteq \Xi, \Xi'$ and $|\Xi, \Xi| \setminus |\Xi, \Xi'| = \emptyset$.

$\Diamond$

Next, we prove (ii) by induction on the derivation of $\Theta \vdash \Phi <: \Psi$.

Base cases: Assume $\Theta \vdash \Phi <: \Psi$ has been derived by ($<:$ reflexivity). Then we have $\Phi = \Psi$, and a proved type $\Theta \vdash \Phi$, where Θ is minimal. Firstly, we know by Proposition 4.45 that for each $(p, \Sigma) \in sta(\Phi) = sta(\Psi)$ type term Σ is a subterm of Φ. Hence, by (i), there exists a type context Θ_Σ such that $\Theta_\Sigma \vdash \Sigma$ is a proved type. Since we can derive proved subtype $\Theta_\Sigma \vdash \Sigma <: \Sigma$ by ($<:$ reflexivity), and due to the fact that addresses are unique in $sta(\Phi)$, i.e. $\big|\{p \mid (p, \Sigma') \in sta(\Phi)\}\big| = \big|sta(\Phi)\big|$ (which we get by inspection of the definition of function sta), we may conclude that (ii) holds in this case.

Suppose $\Theta \vdash \Phi <: \Psi$ has been derived by (Top supertype), i.e. $\Psi = Top$. Then $sta(Top) = \{(\varepsilon, Top)\}$, and there thus is only one pair $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in sta(\Phi) \times sta(Top)$, namely $\big((\varepsilon, \Phi), (\varepsilon, Top)\big)$, since addresses in $sta(\Phi)$ are unique. And for this pair, we get $\Theta \vdash \Phi <: Top$ by (Top supertype), where moreover $\Theta \sqsubseteq \Theta$ holds. Hence, (ii.2) applies, since $|\varepsilon|_1 = |\varepsilon|_7 = 0$.

Induction cases: Assume $\Theta \vdash \Phi <: \Psi$ has been derived by

$$\frac{\Theta \vdash \Phi' <: \Psi}{\Theta \vdash !\Phi' <: \Psi}\ (!\text{ left})\ .$$

By definition of function sta, we then get

$$sta(!\Phi') = \big(sta(\Phi') \setminus \{(\varepsilon, \Phi')\}\big) \cup \{(\varepsilon, !\Phi')\}\ ,$$

and thus induction yields there is a type context $\widehat{\Theta}$ for each pair $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in sta(!\Phi') \times sta(\Psi)$ with $p \neq \varepsilon$ such that $\Theta \sqsubseteq \widehat{\Theta}$ and (ii.2) and (ii.3) hold. The way how $\Theta \vdash !\Phi' <: \Psi$ has been derived shows that for pair $\big((\varepsilon, !\Phi'), (\varepsilon, \Psi)\big)$ we get type context Θ with $\Theta \sqsubseteq \Theta$ such that $\Theta \vdash !\Phi' <: \Psi$ is a proved subtype, and thus (ii.2) applies.
The case where $\Theta \vdash \Phi <: \Psi$ has been derived by (! right) is analogous.

Suppose $\Theta \vdash \Phi <: \Psi$ has been derived by

$$\frac{\Theta \vdash \Psi_1 <: \Phi_1 \qquad \Theta \vdash \Phi_2 <: \Psi_2}{\Theta \vdash (\Phi_1 \multimap \Phi_2) <: (\Psi_1 \multimap \Psi_2)}\ (\text{function subtype})\ .$$

Then we have

$$\begin{aligned}
sta\big((\Phi_1 \multimap \Phi_2)\big) = {} & \big\{(1p'', \Sigma_{\Phi_1}) \mid (p'', \Sigma_{\Phi_1}) \in sta(\Phi_1)\big\} && \big\} =: s_{\Phi_1} \\
& \cup \big\{(2p', \Sigma_{\Phi_2}) \mid (p', \Sigma_{\Phi_2}) \in sta(\Phi_2)\big\} && \big\} =: s_{\Phi_2} \\
& \cup \big\{\big(\varepsilon, (\Phi_1 \multimap \Phi_2)\big)\big\} && \big\} =: s_{\Phi} \\
sta\big((\Psi_1 \multimap \Psi_2)\big) = {} & \big\{(1p', \Sigma_{\Psi_1}) \mid (p', \Sigma_{\Psi_1}) \in sta(\Psi_1)\big\} && \big\} =: s_{\Psi_1} \\
& \cup \big\{(2p'', \Sigma_{\Psi_2}) \mid (p'', \Sigma_{\Psi_2}) \in sta(\Psi_2)\big\} && \big\} =: s_{\Psi_2} \\
& \cup \big\{\big(\varepsilon, (\Psi_1 \multimap \Psi_2)\big)\big\} && \big\} =: s_{\Psi}
\end{aligned}$$

by definition of function sta. By induction, there exists a type context $\widehat{\Theta}$ for each pair $\big((p, \Sigma_1), (p, \Sigma_2)\big) \in \big(sta(\Psi_1) \times sta(\Phi_1)\big) \cup \big(sta(\Phi_2) \times sta(\Psi_2)\big)$ such that $\Theta \sqsubseteq \widehat{\Theta}$ and (ii.2) and (ii.3) hold. To make sure that (ii.2) and (ii.3) extend to all pairs in $sta\big((\Phi_1 \multimap \Phi_2)\big) \times sta\big((\Psi_1 \multimap \Psi_2)\big)$, we need to consider the three subsets $s_{\Phi_1} \times s_{\Psi_1}$ and $s_{\Phi_2} \times s_{\Psi_2}$ and $s_\Phi \times s_\Psi$ (all other pairs that are contained in different combinations of binary Cartesian products of $s_\Phi, s_{\Phi_1}, s_{\Phi_2}, s_\Psi, s_{\Psi_1}, s_{\Psi_2}$ differ in their addresses and thus need not be considered):

($s_{\Phi_1} \times s_{\Psi_1}$): As stated above, by induction, there exists a type context $\widehat{\Theta}$ for each pair $\big((p, \Sigma_{\Psi_1}), (p, \Sigma_{\Phi_1})\big) \in \big(sta(\Psi_1) \times sta(\Phi_1)\big)$ such that $\Theta \sqsubseteq \widehat{\Theta}$ and (ii.2) and (ii.3) hold, i.e.

(ii.2): $\widehat{\Theta} \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ is a proved subtype, if $|p|_1 + |p|_7 \mod 2 = 0$,

(ii.3): $\widehat{\Theta} \vdash \Sigma_{\Phi_1} <: \Sigma_{\Psi_1}$ is a proved subtype, if $|p|_1 + |p|_7 \mod 2 = 1$.

This, however, leads us to

- $\widehat{\Theta} \vdash \Sigma_{\Phi_1} <: \Sigma_{\Psi_1}$ is a proved subtype, if $|1p|_1 + |1p|_7 \mod 2 = 0$,

- $\widehat{\Theta} \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ is a proved subtype, if $|1p|_1 + |1p|_7 \mod 2 = 1$,

where the addresses get an additional 1 in front, and thus $|1p|_1 + |1p|_7 \mod 2 = 1 - (|p|_1 + |p|_7 \mod 2)$. But then, (ii.2) and (ii.3) hold with the above $\widehat{\Theta}$s for each pair $\big((p, \Sigma_{\Phi_1}), (p, \Sigma_{\Psi_1})\big) \in s_{\Phi_1} \times s_{\Psi_1}$, since

$$s_{\Phi_1} \times s_{\Psi_1} = \big\{ \big((1p'', \Sigma_{\Phi_1}), (1p', \Sigma_{\Psi_1})\big) \;\big|\; \big((p'', \Sigma_{\Phi_1}), (p', \Sigma_{\Psi_1})\big) \in sta(\Phi_1) \times sta(\Psi_1) \big\} .$$

$(s_{\Phi_2} \times s_{\Psi_2})$: Since, by induction, (ii.2) and (ii.3) hold for all $\big((p, \Sigma_{\Phi_2}), (p, \Sigma_{\Psi_2})\big) \in \big(sta(\Phi_2) \times sta(\Psi_2)\big)$ and a type context $\widehat{\Theta}$ with $\Theta \sqsubseteq \widehat{\Theta}$ for each such pair, (ii.2) and (ii.3) then also hold for all pairs $\big((2p, \Sigma_{\Phi_2}), (2p, \Sigma_{\Psi_2})\big) \in s_{\Phi_2} \times s_{\Psi_2}$ using the same type contexts, because of $|2p|_1 + |2p|_7 = |p|_1 + |p|_7$.

$(s_\Phi \times s_\Psi)$: Since the only pair in this subset is $\big((\varepsilon, (\Phi_1 \multimap \Phi_2)), (\varepsilon, (\Psi_1 \multimap \Psi_2))\big)$, and since we already have derived proved subtype $\Theta \vdash (\Phi_1 \multimap \Phi_2) <: (\Psi_1 \multimap \Psi_2)$ (where $\Theta \sqsubseteq \Theta$ obviously holds), and because of $|\varepsilon|_1 = |\varepsilon|_7 = 0$, we know that (ii.2) applies here.

The cases where $\Theta \vdash \Phi <: \Psi$ has been derived by one of the rules (polymorphic subtype), (product subtype) and (sum subtype) can be covered analogously. However, the latter two cases are even easier to establish, since there the lines of argument for subsets $s_{\Phi_1} \times s_{\Psi_1}$ and $s_{\Phi_2} \times s_{\Psi_2}$ are both analogous to the above case of $s_{\Phi_2} \times s_{\Psi_2}$.

$\Diamond$

We show (iii) by induction on the length of address $p \in addr_\Phi \cap addr_\Psi$.

Base case: Let $|p| = 0$, i.e. $p = \varepsilon$. By definition of function *sta*, we know $(\varepsilon, \Sigma) \in sta(\Sigma)$ holds for all $\Sigma \in \mathcal{T}_{type}$. Hence, $\varepsilon \in addr_\Upsilon$ trivially holds.

Induction case: Let $p = p'k$ for some $k \in \{1, \ldots, 8\}$ and $p' \in \{1, \ldots, 8\}^*$ with $|p'|_1 + |p'|_7 \mod 2 = 0$. We continue by case distinction over k.

$k = 1$ or $k = 2$. At address p' there must have been a subterm $\Sigma_\Psi = !^n(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ in Ψ, i.e. $(p', !^n(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})) \in sta(\Psi)$. By induction we then know $p' \in addr_\Upsilon$, which implies that there exists a type term Σ_Υ with $(p', \Sigma_\Upsilon) \in sta(\Upsilon)$. Thus, (ii) entails that there exists some type context $\widehat{\Theta}_{\Upsilon\Psi}$ with $\Theta_{\Upsilon\Psi} \sqsubseteq \widehat{\Theta}_{\Upsilon\Psi}$ such that $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Upsilon <: !^n(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ is a proved subtype.
There are two possibilities how this could have been derived: either by iterated application of rules (! left) and (! right) starting from some proved subtype $\widehat{\Theta}_{\Upsilon\Psi} \vdash !^l(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2}) <: !^l(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ which then again has been derived by ($<:$ reflexivity). This entails $\Sigma_\Upsilon = !^{l+m}(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$, where $l, m \geq 0$. Then we clearly have $\{p'1, p'2\} \subseteq addr_\Upsilon$, by definition of function *sta*.
The other possibility is that $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Upsilon <: !^n(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ has been derived by iterated application of rules (! left) and (! right) starting from some proved subtype $\widehat{\Theta}_{\Upsilon\Psi} \vdash (\Sigma_{\Upsilon_1} \multimap \Sigma_{\Upsilon_2}) <: (\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$. This implies that Σ_Υ has the form $\Sigma_\Upsilon = !^m(\Sigma_{\Upsilon_1} \multimap \Sigma_{\Upsilon_2})$ with $m \geq 0$. But then we also have $\{p'1, p'2\} \subseteq addr_\Upsilon$ by definition of function *sta*.

The cases ($k = 3$ or $k = 4$), ($k = 5$ or $k = 6$), ($k = 7$ or $k = 8$) can be solved in the same spirit, while they are connected with $\Sigma_\Psi = !^n(\Sigma_{\Psi_1} \otimes \Sigma_{\Psi_2})$, $\Sigma_\Psi = !^n(\Sigma_{\Psi_1} \oplus \Sigma_{\Psi_2})$, $\Sigma_\Psi = !^n(\forall X{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2})$, respectively.

Now consider the case of $p = p'k$ for some $k \in \{1, \ldots, 8\}$ and $p' \in \{1, \ldots, 8\}^*$ with $|p'|_1 + |p'|_7 \mod 2 = 1$. Here we can argue along similar lines as in the above case. However, we need to work with Σ_Φ from $(p', \Sigma_\Phi) \in sta(\Phi)$ instead of Σ_Ψ, since (ii.3) applies in case of $|p'|_1 + |p'|_7 \mod 2 = 1$. (It does not help much to consider a subtype Σ_Ψ of Σ_Υ, because it does not deliver sufficient information about the structure of Σ_Υ. Since Σ_Φ is supertype of Σ_Υ in this case, its form reveals the required information about the shape of Σ_Υ.)

$\Diamond$

Let us now treat (iv).

At first, we take a look at the existence (and construction) of type context $\widehat{\Theta}_*$, which shall be consistent. By assumption, we have proved type $\Theta_\Phi \vdash \Phi$ with consistent minimal type context Θ_Φ. Hence, by

(i), there exists a type context Θ_{Σ_Φ} with $\Theta_\Phi \sqsubseteq \Theta_{\Sigma_\Phi}$ such that $\Theta_{\Sigma_\Phi} \vdash \Sigma_\Phi$ is a proved type for each $\Sigma_\Phi \in \mathit{subt}(\Phi)$ (and thus for each Σ_Φ with $(p, \Sigma_\Phi) \in \mathit{sta}(\Phi)$, by Proposition 4.45).

We define type context $\widehat{\Theta}_\Phi := \Theta_\Phi, X_1{<:}\Phi_{X_1}, \ldots, X_k{<:}\Phi_{X_k}$, where we collect *all* (and only) the X_i which are bound by type abstractions $(\forall X_i{<:}\Phi_{X_i}.\Sigma_{X_i})$ in Φ. (Please recall we assumed each $Y \in \mathcal{V}_{type}$ is bound at most once in Φ.) We furthermore require for all i with $1 \leq i \leq k$ none of the X_j with $j \leq i$ is bound by type abstraction in Σ_{X_i}. This means, the $X_1, \ldots, X_k$ appear (left of $<:$) in an outermost to innermost order (viewed from left to right) in Θ_Φ when their respective type abstractions are nested in Φ. For type abstractions which are not nested[38] in Φ, we make no assumption about their order in $\widehat{\Theta}_\Phi$.

This construction implies two properties of $\widehat{\Theta}_\Phi$:

- $\widehat{\Theta}_\Phi$ is consistent and
- $\Theta_{\Sigma_\Phi}(Y) = \widehat{\Theta}_\Phi(Y)$ for all of the above mentioned Θ_{Σ_Φ} and for all $Y \in |\Theta_{\Sigma_\Phi}|$.

The second property is an immediate consequence of (i.1) and (i.3). Consistency of $\widehat{\Theta}_\Phi$ is a consequence of Θ_Φ's consistency (by Lemma 4.18) and the way type abstractions in $\Theta_\Phi \vdash \Phi$ are derived by rules (linear-polymorphic type) and (nonlinear-polymorphic type). Either derivation rule starts from a premise $\Xi, X_i{<:}\Phi_{X_i}, \Xi' \vdash \Sigma_{X_i}$, removes pair $X_i{<:}\Phi_{X_i}$ from the type context and thus ends up with proved type $\Xi, \Xi' \vdash\ !^n(\forall X_i <: \Phi_{X_i}.\Sigma_{X_i})$, where consistency of Ξ, Ξ' is provided by the first premise of the used derivation rule (either directly by $\vdash \Xi, \Xi'$ or indirectly by proved subtype $\Xi, \Xi' \vdash \Phi_{X_i} <: !\mathit{Top}$, which leads to consistency of Ξ, Ξ' due to Lemma 4.19). All other derivation rules leave the involved type contexts unchanged, which implies that pairs $Y{<:}\Phi_Y$ where Y is bound in Σ_{X_i} cannot appear in Ξ, Ξ'. This entails we can derive proved type $\Theta_\Phi, X_1{<:}\Phi_{X_1}, \ldots, X_{i-1}{<:}\Phi_{X_{i-1}} \vdash \Phi_{X_i}$ as long as all type variables Z that appear freely in $(\forall X_i{<:}\Phi_{X_i}.\Sigma_{X_i})$ have a pair $Z{<:}\Phi_Z$ in $\Theta_\Phi, X_1{<:}\Phi_{X_1}, \ldots, X_{i-1}{<:}\Phi_{X_{i-1}}$. But this is exactly what we guarantee by construction of $\widehat{\Theta}_\Phi$. Analogously to this, we define type context $\widehat{\Theta}_\Psi$ based on proved type $\Theta_\Psi \vdash \Psi$.

We base the definition of $\widehat{\Theta}_*$ on type contexts $\widehat{\Theta}_\Phi$ and $\widehat{\Theta}_\Phi$ as follows:
for all $Y \in \mathcal{V}_{type}$ we set

$$\widehat{\Theta}_*(Y) := \begin{cases} \widehat{\Theta}_\Phi(Y) & \text{if } \widehat{\Theta}_\Psi(Y) = \bot \text{ and } \widehat{\Theta}_\Phi(Y) \neq \bot, \\ \widehat{\Theta}_\Psi(Y) & \text{if } \widehat{\Theta}_\Phi(Y) = \bot \text{ and } \widehat{\Theta}_\Psi(Y) \neq \bot, \\ \widehat{\Theta}_\Psi(Y) & \text{if } \big((p, (\forall Y{<:}\Sigma_{\Phi_1}.\Sigma_{\Phi_2})), (p, (\forall Y{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2}))\big) \in \mathit{sta}(\Phi) \times \mathit{sta}(\Psi) \\ & \text{and } |p|_1 + |p|_7 \mod 2 = 0, \\ \widehat{\Theta}_\Phi(Y) & \text{if } \big((p, (\forall Y{<:}\Sigma_{\Phi_1}.\Sigma_{\Phi_2})), (p, (\forall Y{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2}))\big) \in \mathit{sta}(\Phi) \times \mathit{sta}(\Psi) \\ & \text{and } |p|_1 + |p|_7 \mod 2 = 1, \\ \Theta(Y) & \text{otherwise.} \end{cases}$$

(This definition assumes that there are no type abstractions in Φ that bind the same type variable as a type abstraction in Ψ which has a different address than the one in Φ. This assumption in not a problem, if we use α-equivalence to circumvent such complications.
The last line "$\Theta(Y)$ otherwise" in the above definition takes care of all free type variables that appear in Φ *and* in Ψ. For all variables Z that do not appear in Φ or Ψ (neither freely nor bound), we have $\Theta(Z) = \bot$ due to our assumption $|\Theta| = |\Theta_\Phi| \cup |\Theta_\Psi|$.)

Concerning the order of variable-type pairs in $\widehat{\Theta}_*$, the following shall hold:

$$\widehat{\Theta}_\Phi|_{\{Y \mid \widehat{\Theta}_\Phi(Y) = \widehat{\Theta}_*(Y)\}} \sqsubseteq \widehat{\Theta}_* \quad \text{and} \quad \widehat{\Theta}_\Psi|_{\{Y \mid \widehat{\Theta}_\Psi(Y) = \widehat{\Theta}_*(Y)\}} \sqsubseteq \widehat{\Theta}_* \ .$$

To obtain a consistent $\widehat{\Theta}_*$ which matches the above specification and this constraint on its inner order, we start from proved type $\Theta_\Phi \vdash \Phi$. Due to the definition of $\widehat{\Theta}_\Phi$, we know $\Theta_\Phi \sqsubseteq \widehat{\Theta}_\Phi$ holds and we already concluded $\widehat{\Theta}_\Phi$ is consistent. Using rule (type weakening) we can thus derive proved type $\widehat{\Theta}_\Phi \vdash \Phi$. In an analogous way, we obtain proved type $\widehat{\Theta}_\Psi \vdash \Psi$.

Consider type variables $X \in |\widehat{\Theta}_\Phi| \cap |\widehat{\Theta}_\Psi|$. If they appear as free type variables in Φ and in Ψ, then we have $X \in \mathit{ftyv}(\Phi) \cap \mathit{ftyv}(\Psi) \subseteq |\Theta_\Phi| \cap |\Theta_\Psi| \subseteq |\Theta|$, which entails $\widehat{\Theta}_\Phi(X) = \widehat{\Theta}_\Psi(X)$, since we assume

[38] An example for nested type abstractions is $\Theta \vdash (\forall X_1 <: (\forall X_2 <: \Phi_{X_2}.\Psi_2).\Psi_1)$ and an example for non-nested type abstractions is $\Theta \vdash ((\forall X_1{<:}\Phi_{X_1}.\Psi_1) \multimap (\forall X_2{<:}\Phi_{X_2}.\Psi_2))$.

$\Theta_\Phi \sqsubseteq \Theta$ and $\Theta_\Psi \sqsubseteq \Theta$. For type variables X that appear bound in both type terms Φ and Ψ, i.e. the ones with

$$\big((p, (\forall X{<:}\Sigma_{\Phi_1}.\Sigma_{\Phi_2})), (p, (\forall X{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2}))\big) \in sta(\Phi) \times sta(\Psi) \ ,$$

we defined $\widehat{\Theta}_*$ such that

$$\widehat{\Theta}_*(X) = \begin{cases} \widehat{\Theta}_\Psi(X) & \text{if } |p|_1 + |p|_7 \mod 2 = 0, \\ \widehat{\Theta}_\Phi(X) & \text{if } |p|_1 + |p|_7 \mod 2 = 1. \end{cases}$$

What we additionally get from proved subtypes $\Theta_{\Phi\Upsilon} \vdash \Phi <: \Upsilon$ and $\Theta_{\Upsilon\Psi} \vdash \Upsilon <: \Psi$, however, is the following chain of conclusions:

- by (iii), we have $\big((p7, \Sigma_{\Phi_1}), (p7, \Sigma_{\Upsilon_1})\big) \in sta(\Phi) \times sta(\Upsilon)$ and $\big((p7, \Sigma_{\Upsilon_1}), (p7, \Sigma_{\Psi_1})\big) \in sta(\Upsilon) \times sta(\Psi)$;
- by (ii.2) and (ii.3), there exist type contexts $\widehat{\Theta}_{\Phi\Upsilon}$ and $\widehat{\Theta}_{\Upsilon\Psi}$ such that
  - $\widehat{\Theta}_{\Phi\Upsilon} \vdash \Sigma_{\Upsilon_1} <: \Sigma_{\Phi_1}$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_{\Psi_1} <: \Sigma_{\Upsilon_1}$ are proved types, if $|p|_1 + |p|_7 \mod 2 = 0$, and
  - $\widehat{\Theta}_{\Phi\Upsilon} \vdash \Sigma_{\Phi_1} <: \Sigma_{\Upsilon_1}$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_{\Upsilon_1} <: \Sigma_{\Psi_1}$ are proved types, if $|p|_1 + |p|_7 \mod 2 = 1$;
- by Proposition 4.16 and transitivity of the subtype relation (Proposition 4.8), we get
  - $\Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$, if $|p|_1 + |p|_7 \mod 2 = 0$, and
  - $\Sigma_{\Phi_1} <: \Sigma_{\Psi_1}$, if $|p|_1 + |p|_7 \mod 2 = 1$;
- by definition of $\widehat{\Theta}_*$ and by $\widehat{\Theta}_\Phi(Y) = \Sigma_{\Phi_1}$ and $\widehat{\Theta}_\Psi(Y) = \Sigma_{\Psi_1}$ (which hold by construction of $\widehat{\Theta}_\Phi$ and $\widehat{\Theta}_\Psi$), we know
  - $\widehat{\Theta}_*(Y) = \Sigma_{\Psi_1} <: \Sigma_{\Phi_1} = \widehat{\Theta}_\Phi(Y)$, if $|p|_1 + |p|_7 \mod 2 = 0$, and
  - $\widehat{\Theta}_*(Y) = \Sigma_{\Phi_1} <: \Sigma_{\Psi_1} = \widehat{\Theta}_\Psi(Y)$, if $|p|_1 + |p|_7 \mod 2 = 1$;
- by reflexivity of the subtype relation (axiom (1) for $<:$), we thus have $\widehat{\Theta}_*(Y) <: \widehat{\Theta}_\Phi(Y)$ and $\widehat{\Theta}_*(Y) <: \widehat{\Theta}_\Psi(Y)$, independent of how many 1s and 7s occur in p.

The bottom line of all this is due to Lemma 4.9(i), which then entails that if the type bound of variable Y has a leading exponential in $\widehat{\Theta}_\Phi$ or in $\widehat{\Theta}_\Psi$, then so has the type bound assigned to Y in $\widehat{\Theta}_*$. Formally, this amounts to

- $\widehat{\Theta}_\Phi(Y) = !\Phi_Y$ implies $\widehat{\Theta}_*(Y) = !\Phi'_Y$ and
- $\widehat{\Theta}_\Psi(Y) = !\Phi_Y$ implies $\widehat{\Theta}_*(Y) = !\Phi'_Y$ for some type term Φ'_Y.

Having gained this knowledge, we can now employ rule (type bound replacement) to establish a consistent $\widehat{\Theta}_*$. Recall we already have obtained proved types $\widehat{\Theta}_\Phi \vdash \Phi$ and $\widehat{\Theta}_\Psi \vdash \Psi$ above, and also recall $\widehat{\Theta}_\Phi$ and $\widehat{\Theta}_\Psi$ are of the form $\widehat{\Theta}_\Phi = \Theta_\Phi, \Xi_\Phi$ and $\widehat{\Theta}_\Psi = \Theta_\Psi, \Xi_\Psi$. Since we know

- $\Theta_\Phi \sqsubseteq \widehat{\Theta}_\Phi$ and $\Theta_\Phi \sqsubseteq \Theta$,
- $\Theta_\Psi \sqsubseteq \widehat{\Theta}_\Psi$ and $\Theta_\Psi \sqsubseteq \Theta$, and
- Θ is consistent,

we may weaken proved types $\Theta_\Phi, \Xi_\Phi \vdash \Phi$ and $\Theta_\Psi, \Xi_\Psi \vdash \Psi$ to obtain proved types $\Theta, \Xi_\Phi \vdash \Phi$ and $\Theta, \Xi_\Psi \vdash \Psi$, where we may, if necessary, rearrange variable-type pairs $Y{<:}\Phi_Y$ with $Y \in |\Xi_\Phi| \cap |\Xi_\Psi|$ in Ξ_Φ and Ξ_Ψ in a consistency preserving way (outermost-bound to innermost-bound from left to right) so that they appear in the same order in both type contexts, i.e. if $Y{<:}\Phi_Y$ appears left of $Z{<:}\Phi_Z$ in Ξ_Φ (possibly with type-variable pairs between them), so does $Y{<:}\Phi_Y$ appear left of $Z{<:}\Phi_Z$ in Ξ_Ψ.[39]

Assume (without loss of generality, since the other case is symmetric) we have $\widehat{\Theta}_*(X) = \widehat{\Theta}_\Psi(X)$ for the leftmost variable-type pair $X{<:}\Xi_\Phi(X)$ in Ξ_Φ with $\Xi_\Phi(X) \neq \Xi_\Psi(X)$. If there are pairs $Y_1{<:}\Xi_\Psi(Y_1), \ldots, Y_k{<:}\Xi_\Psi(Y_k)$ in front of $X{<:}\Xi_\Psi(X)$ in Ξ_Ψ with $\{Y_1, \ldots, Y_k\} \cap |\Xi_\Phi| = \emptyset$, then we may weaken

[39] Please note this is possible, since we can rename bound variables in case of any conflicts.

Θ, Ξ_Φ to include them, and get the result $\Theta, \Xi, X{<:}\Xi_\Phi(X), \Xi'_\Phi$. Dually, we weaken Θ, Ξ_Ψ to contain all pairs between Θ and $X{<:}\Xi_\Phi(X)$ that do not yet occur in Θ, Ξ_Ψ, and thus obtain $\Theta, \Xi, X{<:}\Xi_\Psi(X), \Xi'_\Psi$. By consistency of $\Theta, \Xi, X{<:}\Xi_\Psi(X), \Xi'_\Psi$, we then know $\Theta, \Xi \vdash \Xi_\Psi(X)$ is a proved type, and we furthermore deduced above that $\Xi_\Psi(X) = \widehat{\Theta}_*(X)$ has a leading exponential if $\Xi_\Phi(X)$ has. Hence, we may apply (type bound replacement) to derive proved type $\Theta, \Xi, X{<:}\widehat{\Theta}_*(X), \Xi'_\Phi \vdash \Phi$. Since we already have proved type $\Theta, \Xi, X{<:}\widehat{\Theta}_*(X), \Xi'_\Psi \vdash \Psi$, we came one step closer to our goal.

To clarify the just described procedure, we give a short visualization:

$$
\begin{array}{lll}
\Theta_\Phi, \Xi_\Phi \vdash \Phi & \text{(type weakening)} & \Theta, \Xi_\Phi \vdash \Phi \qquad\qquad \overbrace{\Theta, \Xi''_\Phi}^{\sqsubseteq \widehat{\Theta}_*}, X{<:}\overbrace{\Xi_\Phi(X)}^{\neq \Xi_\Psi(X)}, \Xi'_\Phi \vdash \Phi \\
\Theta_\Psi, \Xi_\Psi \vdash \Psi & \rightsquigarrow & \Theta, \Xi_\Psi \vdash \Psi \qquad\qquad = \qquad \underbrace{\Theta, \Xi''_\Psi}_{\sqsubseteq \widehat{\Theta}_*}, X{<:}\underbrace{\Xi_\Psi(X)}_{= \widehat{\Theta}_*(X)}, \Xi'_\Psi \vdash \Psi
\end{array}
$$

$$
\begin{array}{lll}
 & \text{(type weakening)} & \Theta, \Xi, X{<:}\overbrace{\Xi_\Phi(X)}^{\neq \Xi_\Psi(X)}, \Xi'_\Phi \vdash \Phi \\
 & \rightsquigarrow & \Theta, \Xi, X{<:}\underbrace{\Xi_\Psi(X)}_{= \widehat{\Theta}_*(X)}, \Xi'_\Psi \vdash \Psi \\
 & \text{(type bound replacement)} & \overbrace{\Theta, \Xi, X{<:}\widehat{\Theta}_*(X)}^{\sqsubseteq \widehat{\Theta}_*}, \Xi'_\Phi \vdash \Phi \\
 & \rightsquigarrow & \underbrace{\Theta, \Xi, X{<:}\widehat{\Theta}_*(X)}_{\sqsubseteq \widehat{\Theta}_*}, \Xi'_\Psi \vdash \Psi \\
 & \rightsquigarrow \quad \ldots &
\end{array}
$$

Continuing in this way (i.e. for the leftmost variable-type pair $X{<:}\Phi_X$ in Ξ'_Φ or Ξ'_Ψ with $\Xi'_\Phi(X) \neq \Xi'_\Psi(X)$), we can eventually derive proved types $\widehat{\Theta}_* \vdash \Phi$ and $\widehat{\Theta}_* \vdash \Psi$, which finally confirms consistency of type context $\widehat{\Theta}_*$ (using Lemma 4.18).

Next, consider set $addr_\Phi \cap addr_\Psi$ (where sets $addr_\Phi$ and similar ones are defined as in (iii)). Since type terms are finite, all addresses in this set are finite, and each address $p \in \{1, \ldots, 8\}^*$ can be assigned a length, which we define as the usual word length, denoted $|p|$ with $|p| \in \mathbb{N}$. Then we can partition the set $addr_\Phi \cap addr_\Psi$ into disjoint subsets $A_0, \ldots, A_{l_{\max}} \subseteq addr_\Phi \cap addr_\Psi$, where each A_i contains exactly the addresses of length i with $0 \leq i \leq l_{\max}$, and where $l_{\max}$ shall be the maximal length of an address in $addr_\Phi \cap addr_\Psi$.

We start with an arbitrary address $p \in A_{l_{\max}}$. Since there are no longer addresses than p in $addr_\Phi \cap addr_\Psi$, at least one of the Σ_Φ, Σ_Ψ with $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in sta(\Phi) \times sta(\Psi)$ is either a type constant $!^n\, Top$, $!^n\, Unit$ or $!^n\, Qbit$ (possibly equipped with $n \geq 0$ leading exponentials), or it is a type variable $!^n Y$ (also possibly equipped with leading exponentials).[40]

We distinguish two cases:

$|p|_1 + |p|_7 = 0 \mod 2$. By (iii) we have $p \in addr_\Upsilon$, and thus $(p, \Sigma_\Upsilon) \in sta(\Upsilon)$. Hence, by (ii), there exist type contexts $\widehat{\Theta}_{\Phi\Upsilon}$ and $\widehat{\Theta}_{\Upsilon\Psi}$ such that $\widehat{\Theta}_{\Phi\Upsilon} \vdash \Sigma_\Phi <: \Sigma_\Upsilon$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Upsilon <: \Sigma_\Psi$ are proved subtypes.

Assume

$$\Sigma_\Phi = !^n \Sigma'_\Phi \in \big\{!^{n'} Top, !^{n'} Unit, !^{n'} Qbit \;\big|\; n' \geq 0\big\} \cup \big\{!^{n'} Y \;\big|\; Y \in \mathcal{V}_{type}, n' \geq 0\big\}$$

for linear Σ'_Φ. By inspection of the derivation rules for proved subtypes, we get from $\widehat{\Theta}_{\Phi\Upsilon} \vdash !^n\Sigma'_\Phi <: \Sigma_\Upsilon$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Upsilon <: \Sigma_\Psi$ that either $\Sigma_\Psi = !^m \Sigma'_\Phi$ or $\Sigma_\Psi = !^m\, Top$ holds, where $m = 0$ if $n = 0$ and $m \geq 0$ otherwise. By (i), we know that $\widehat{\Theta}_{\Sigma_\Phi} \vdash !^n\Sigma'_\Phi$ is a proved type for some type context

[40] This holds by definition of function *sta*, since otherwise we would either have longer addresses in $addr_\Phi \cap addr_\Psi$, or (ii) or (iii) would be violated for $\Theta_{\Phi\Upsilon} \vdash \Phi <: \Upsilon$ or $\Theta_{\Upsilon\Psi} \vdash \Upsilon <: \Psi$.

$\widehat{\Theta}_{\Sigma_\Phi}$ with $\Theta_\Phi \sqsubseteq \widehat{\Theta}_{\Sigma_\Phi}$, and where all $Y \in |\widehat{\Theta}_{\Sigma_\Phi}| \setminus |\Theta_\Phi|$ appear bound in Φ with assigned type bound $\widehat{\Theta}_{\Sigma_\Phi}(Y)$. By construction of $\widehat{\Theta}_\Phi$, we can, if necessary, permute $\widehat{\Theta}_{\Sigma_\Phi}$ in a consistency preserving way so that $\widehat{\Theta}_{\Sigma_\Phi} \sqsubseteq \widehat{\Theta}_\Phi$ holds. Consistency of $\widehat{\Theta}_\Phi$ has already been shown above. Thus, by (type weakening), we obtain proved type $\widehat{\Theta}_\Phi \vdash !^n\Sigma'_\Phi$. When we have been showing consistency of $\widehat{\Theta}_*$ above, we derived proved type $\widehat{\Theta}_* \vdash \Phi$ from $\widehat{\Theta}_\Phi \vdash \Phi$. Applying the same strategy (and perhaps an additional weakening step at the end), we can now derive $\widehat{\Theta}_* \vdash !^n\Sigma'_\Phi$ from $\widehat{\Theta}_\Phi \vdash !^n\Sigma'_\Phi$.[41]

In case $\Sigma_\Psi = !^m\Sigma'_\Phi$ we can derive proved type $\widehat{\Theta}_* \vdash !^{\min(n,m)}\Sigma'_\Phi$ by (! elimination) from $\widehat{\Theta}_* \vdash !^n\Sigma'_\Phi$ or by doing nothing at all (where we keep in mind that $n = 0$ entails $m = 0$). Furthermore, we may then derive proved subtype $\widehat{\Theta}_* \vdash !^{\min(n,m)}\Sigma'_\Phi <: !^{\min(n,m)}\Sigma'_\Phi$ by ($<:$ reflexivity), and after this, we use rules (! left) and (! right) an appropriate number of times to finally obtain proved subtype $\widehat{\Theta}_* \vdash !^n\Sigma'_\Phi <: !^m\Sigma'_\Phi = \widehat{\Theta}_* \vdash \Sigma_\Phi <: \Sigma_\Psi$.
In case $\Sigma_\Psi = !^m\mathit{Top}$ we can derive proved subtype $\widehat{\Theta}_* \vdash !^n\Sigma'_\Phi <: \mathit{Top}$ by (Top supertype), and after this, we use rule (! right) the appropriate number of times to finally obtain proved subtype $\widehat{\Theta}_* \vdash !^n\Sigma'_\Phi <: !^m\mathit{Top} = \widehat{\Theta}_* \vdash \Sigma_\Phi <: \Sigma_\Psi$.
In any of the two above cases, it might be possible to reduce $\widehat{\Theta}_*$ in size using Corollary 4.28(i), and still fulfill (iv.2.1), (iv.2.2) and (iv.2.3). Hence, we define $\widehat{\Theta}$ to be the smallest type context such that $\Theta \sqsubseteq \widehat{\Theta} \sqsubseteq \widehat{\Theta}_*$ holds and $\widehat{\Theta} \vdash \Sigma_\Phi <: \Sigma_\Psi$ is derivable.

Assume

$$\Sigma_\Psi = !^n\Sigma'_\Psi \in \left\{!^{n'}\mathit{Unit}, !^{n'}\mathit{Qbit} \mid n' \geq 0\right\} \cup \left\{!^{n'}Y \mid Y \in \mathcal{V}_{\mathit{type}}, n' \geq 0\right\}$$

for linear Σ'_Ψ. By inspection of the derivation rules, we then get from $\widehat{\Theta}_{\Phi\Upsilon} \vdash \Sigma_\Phi <: \Sigma_\Upsilon$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Upsilon <: \Sigma_\Psi$ that $\Sigma_\Phi = !^m\Sigma'_\Psi$ holds, where we have again $m = 0$ if $n = 0$ and $m \geq 0$ otherwise. We have already shown above, how to continue to finally obtain proved subtype $\widehat{\Theta} \vdash \Sigma_\Phi <: \Sigma_\Psi$ with $\Theta \sqsubseteq \widehat{\Theta} \sqsubseteq \widehat{\Theta}_*$.

Assume $\Sigma_\Psi = !^n\mathit{Top}$. Then we know nothing about the form of Σ_Φ. However, as already written above, we can derive proved subtype $\widehat{\Theta}_{\Sigma_\Phi} \vdash \Sigma_\Phi <: \mathit{Top}$ by (Top supertype), and eventually obtain $\widehat{\Theta}_{\Sigma_\Phi} \vdash \Sigma_\Phi <: !^n\mathit{Top}$ by applying (! right) an appropriate number of times. How to get $\widehat{\Theta} \vdash \Sigma_\Phi <: \Sigma_\Psi$ with $\Theta \sqsubseteq \widehat{\Theta} \sqsubseteq \widehat{\Theta}_*$ from here is described above.

$|p|_1 + |p|_7 = 1 \bmod 2$. This case is dual to the previous one in the sense that the direction of subtypes is reversed. For instance, (ii) in this case yields the existence of type contexts $\widehat{\Theta}_{\Phi\Upsilon}$ and $\widehat{\Theta}_{\Upsilon\Psi}$ so that we can derive proved subtypes $\widehat{\Theta}_{\Phi\Upsilon} \vdash \Sigma_\Upsilon <: \Sigma_\Phi$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Psi <: \Sigma_\Upsilon$. Hence, we have to replace Φ by Ψ, and vice versa, in a sensible way in the previous arguments to get a valid argument for the present case. In the end, we thus obtain proved subtype $\widehat{\Theta} \vdash \Sigma_\Psi <: \Sigma_\Phi$.

We have now shown that there exists a smallest type context $\widehat{\Theta}$ with $\Theta \sqsubseteq \widehat{\Theta} \sqsubseteq \widehat{\Theta}_*$ for each $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in \mathit{sta}(\Phi) \times \mathit{sta}(\Psi)$ with $p \in A_{l_{\max}}$ such that (iv.2.1), (iv.2.2) and (iv.2.3) are fulfilled.

Next, we take a look at set A_i with $0 \leq i < l_{\max}$ and assume inductively, that we have already covered all pairs $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in \mathit{sta}(\Phi) \times \mathit{sta}(\Psi)$ with $p \in A_{i+1} \cup \ldots \cup A_{l_{\max}}$ and found type contexts $\widehat{\Theta}$ for them, respectively, such that (iv.2.1), (iv.2.2), (iv.2.3) hold. Thus, consider an arbitrary $p \in A_i$. If there is no $pk \in A_{i+1}$ for any $k \in \{1, \ldots, 8\}$, then we are in the same position as for set $A_{l_{\max}}$, and we have already argued above what to do then. Hence, assume there is some such k with $pk \in A_{i+1}$. The definition of function sta tells us there must also be a corresponding $p(k+1)$ or $p(k-1)$ in A_{i+1}.
Again, we distinguish two cases:

$|p|_1 + |p|_7 = 0 \bmod 2$. Now we consider $\big((p, \Sigma_\Phi), (p, \Sigma_\Psi)\big) \in \mathit{sta}(\Phi) \times \mathit{sta}(\Psi)$. Suppose we have found $p1$ and $p2$ in A_{i+1}. Then Σ_Φ and Σ_Ψ are of the form $\Sigma_\Phi = !^n(\Sigma_{\Phi_1} \multimap \Sigma_{\Phi_2})$ and $\Sigma_\Psi = !^m(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ with $n = 0 \Rightarrow m = 0$.[42] We ignore the leading exponentials in the following argument, because

[41] When we have been deriving proved type $\widehat{\Theta}_* \vdash \Phi$ from $\widehat{\Theta}_\Phi \vdash \Phi$, we used consistency of $\widehat{\Theta}_\Psi$ to support the construction. Here, we may use consistency of $\widehat{\Theta}_*$ instead.

[42] Condition $n = 0 \Rightarrow m = 0$ can be concluded from the fact that we have assumed proved subtypes $\Theta_{\Phi\Upsilon} \vdash \Phi <: \Upsilon$ and $\Theta_{\Upsilon\Psi} \vdash \Upsilon <: \Psi$, which entails proved subtypes $\widehat{\Theta}_{\Phi\Upsilon} \vdash \Sigma_\Phi <: \Sigma_\Upsilon$ and $\widehat{\Theta}_{\Upsilon\Psi} \vdash \Sigma_\Upsilon <: \Sigma_\Psi$ are derivable for some type contexts $\widehat{\Theta}_{\Phi\Upsilon}$ and $\widehat{\Theta}_{\Upsilon\Psi}$ due to (ii). By Corollary 4.17, we then know the numbers n, l and m of leading exponentials in Σ_Φ, Σ_Υ and Σ_Ψ, respectively, must obey condition $n = 0 \Rightarrow l = 0 \Rightarrow m = 0$.

once we can derive $\widehat{\Theta} \vdash (\Sigma_{\Phi_1} \multimap \Sigma_{\Phi_2}) <: (\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$, it is then easy to derive $\widehat{\Theta} \vdash !^n(\Sigma_{\Phi_1} \multimap \Sigma_{\Phi_2}) <: !^m(\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ by n applications of (! left) followed by m applications of rule (! right).

Hence, assume Σ_Φ and Σ_Ψ are of the form $\Sigma_\Phi = (\Sigma_{\Phi_1} \multimap \Sigma_{\Phi_2})$ and $\Sigma_\Psi = (\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})$ with

$$\big\{\big((p1, \Sigma_{\Phi_1}), (p1, \Sigma_{\Psi_1})\big), \big((p2, \Sigma_{\Phi_2}), (p2, \Sigma_{\Psi_2})\big)\big\} \subseteq sta(\Phi) \times sta(\Psi) \ .$$

By induction, we then get proved subtypes $\widehat{\Theta}_1 \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ (with a reversed direction regarding the subtype relation, since $|p|_1 + |p|_7 \mod 2 = 1 - |p1|_1 + |p1|_7 \mod 2$) and $\widehat{\Theta}_2 \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}$ with $\Theta \sqsubseteq \widehat{\Theta}_1 \sqsubseteq \widehat{\Theta}_*$ and $\Theta \sqsubseteq \widehat{\Theta}_2 \sqsubseteq \widehat{\Theta}_*$. Now, let $\widehat{\Theta}$ be the smallest type context with $\widehat{\Theta}_1 \sqsubseteq \widehat{\Theta}$ and $\widehat{\Theta}_2 \sqsubseteq \widehat{\Theta}$. Then we immediately know that $\widehat{\Theta} \sqsubseteq \widehat{\Theta}_*$ and that $\widehat{\Theta}$ is consistent, since $\widehat{\Theta}_1$ and $\widehat{\Theta}_2$ are. Having this, we may then apply (subtype weakening) to obtain proved subtypes $\widehat{\Theta} \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ and $\widehat{\Theta} \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}$. But then, we may of course perform derivation

$$\frac{\widehat{\Theta} \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1} \qquad \widehat{\Theta} \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}}{\widehat{\Theta} \vdash (\Sigma_{\Phi_1} \multimap \Sigma_{\Phi_2}) <: (\Sigma_{\Psi_1} \multimap \Sigma_{\Psi_2})}\ \text{(function subtype)} \ .$$

Clearly, the cases where we find $p3$ and $p4$ in A_{i+1} or $p5$ and $p6$ can be handled in the same spirit, since they correspond to

$$\Sigma_\Phi = (\Sigma_{\Phi_1} \otimes \Sigma_{\Phi_2}) \ \text{ and } \ \Sigma_\Psi = (\Sigma_{\Psi_1} \otimes \Sigma_{\Psi_2})$$

or

$$\Sigma_\Phi = (\Sigma_{\Phi_1} \oplus \Sigma_{\Phi_2}) \ \text{ and } \ \Sigma_\Psi = (\Sigma_{\Psi_1} \oplus \Sigma_{\Psi_2}) \ ,$$

respectively. These cases are even a bit easier, since they do not induce a switch in the direction of the subtype relation between Σ_{Φ_1} and Σ_{Ψ_1}.

Suppose we found $p7$ and $p8$ in A_{i+1}. Then Σ_Φ and Σ_Ψ are of the form $\Sigma_\Phi = (\forall X{<:}\Sigma_{\Phi_1}.\Sigma_{\Phi_2})$ and $\Sigma_\Psi = (\forall X{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2})$ with

$$\big\{\big((p7, \Sigma_{\Phi_1}), (p7, \Sigma_{\Psi_1})\big), \big((p8, \Sigma_{\Phi_2}), (p8, \Sigma_{\Psi_2})\big)\big\} \subseteq sta(\Phi) \times sta(\Psi) \ .$$

(Again, we ignore leading exponentials, as we have done in the previous induction step, since we can add them easily as soon as we have derived $\widehat{\Theta} \vdash \Sigma_\Phi <: \Sigma_\Psi$.)

By induction, we get proved subtypes $\widehat{\Theta}_1 \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ (also with a reversed direction regarding the subtype relation due to $|p7|_1 + |p7|_7 \mod 2 = 1 - |p|_1 + |p|_7 \mod 2$) and $\widehat{\Theta}_2 \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}$ with $\Theta \sqsubseteq \widehat{\Theta}_1 \sqsubseteq \widehat{\Theta}_*$ and $\Theta \sqsubseteq \widehat{\Theta}_2 \sqsubseteq \widehat{\Theta}_*$. Since we assumed from the beginning none of the $Y \in |\Theta|$ appears bound in Φ or Ψ and different type abstractions in Φ bind different type variables (and analogously for Ψ), we conclude that in particular $X \notin ftyv(\Sigma_{\Phi_1}) \cup ftyv(\Sigma_{\Psi_1})$. And since $\widehat{\Theta}_1$ is a smallest type context fulfilling (iv.2.1), (iv.2.2) and (iv.2.3), we know $X \notin |\widehat{\Theta}_1|$.

Now let $\widehat{\Xi}, \widehat{\Xi}'$ be the smallest consistent type context fulfilling

- $\widehat{\Theta}_1 \sqsubseteq \widehat{\Xi}, \widehat{\Xi}' \sqsubseteq \widehat{\Theta}_*$ and
- $\widehat{\Theta}_2 \sqsubseteq \widehat{\Xi}, X{<:}\widehat{\Theta}_*(X), \widehat{\Xi}' \sqsubseteq \widehat{\Theta}_*$.

(This can be constructed starting from $\widehat{\Theta}_*$ by removing all variable-type pairs $Y <: \Phi_Y$ that are already bound by type abstractions in Σ_{Φ_1}, Σ_{Φ_1}, Σ_{Ψ_1} and Σ_{Ψ_2}. By construction of $\widehat{\Theta}_*$ – especially due to the outermost-bound-to-innermost-bound order of variable-type pairs – this leads to a consistent type context. The thus gained type context may be minimized even further, if possible.)

Since $\widehat{\Theta}_*$ is constructed so that (iv.1.1) is fulfilled, we then immediately know $\widehat{\Theta}_*(X) = \Sigma_{\Psi_1}$. Having this, we apply (type weakening) to obtain proved subtypes $\widehat{\Xi}, \widehat{\Xi}' \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ from $\widehat{\Theta}_1 \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1}$ and $\widehat{\Xi}, X{<:}\widehat{\Theta}_*(X), \widehat{\Xi}' \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}$ from $\widehat{\Theta}_2 \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}$. Hence, we may finally perform derivation

$$\frac{\widehat{\Xi}, \widehat{\Xi}' \vdash \Sigma_{\Psi_1} <: \Sigma_{\Phi_1} \qquad \widehat{\Xi}, X{<:}\overbrace{\widehat{\Theta}_*(X)}^{= \Sigma_{\Psi_1}}, \widehat{\Xi}' \vdash \Sigma_{\Phi_2} <: \Sigma_{\Psi_2}}{\widehat{\Xi}, \widehat{\Xi}' \vdash (\forall X{<:}\Sigma_{\Phi_1}.\Sigma_{\Phi_2}) <: (\forall X{<:}\Sigma_{\Psi_1}.\Sigma_{\Psi_2})}\ \text{(polymorphic subtype)} \ .$$

$|p|_1 + |p|_7 = 1 \mod 2$. Again, this case is dual to the previous case with respect to the direction of the involved subtyping statements.

□

The just proven lemma widely opens the door to another highlight of the present work. In fact, it is almost itself the highlight. But we have not yet arrived at the peak of our current argument. We have emphasized before that we can only establish a weak transitivity result for proved subtypes, namely given proved subtypes $\Theta \vdash \Phi <: \Upsilon$ and $\Theta \vdash \Upsilon <: \Psi$, we can only guarantee the derivability of $\Theta \vdash \Phi <: \Psi$ if $\Theta \vdash \Phi$ and $\Theta \vdash \Psi$ are proved types. But exactly because of this peculiarity, the just explained weak transitivity then does not give any guarantees in case of a longer chain of proved subtypes, such as $\Theta \vdash \Phi <: \Upsilon_1$, $\Theta \vdash \Upsilon_1 <: \Upsilon_2$, $\Theta \vdash \Upsilon_2 <: \Psi$ or even longer ones. The problem is we do not have any result which entails the derivability of proved types $\Theta \vdash \Upsilon_1$ or $\Theta \vdash \Upsilon_2$ in the described setting. Such a result would immediately entail transitivity also stretches over such chains. We can, however, modify the above lemma to also treat such chains of proved subtypes. We detail on this in the proof of Theorem 4.47. But despite this technical improvement, the main contribution of the theorem is a cosmetic one. We therein reformulate the heavily technically flavored item (iv) in the previous lemma into a form which concentrates on transitivity, is relieved of the numerous technical conditions and is thus much more intuitive to use and grasp in its essence.

Theorem 4.47. *Let $\Theta \vdash \Phi$ and $\Theta \vdash \Psi$ be proved types. If there exists a sequence*

$$\Theta_1 \vdash \Upsilon_0 <: \Upsilon_1, \ldots, \Theta_{i+1} \vdash \Upsilon_i <: \Upsilon_{i+1}, \ldots, \Theta_k \vdash \Upsilon_{k-1} <: \Upsilon_k \quad \text{with } 1 \leq i < k \text{ and } k \geq 1$$

of proved subtypes, where we set $\Upsilon_0 = \Phi$ and $\Upsilon_k = \Psi$, then we can derive proved subtype $\Theta \vdash \Phi <: \Psi$.

Proof.

The case of $k = 1$ looks trivial at first glance, since then we already have proved subtype $\Theta_1 \vdash \Phi <: \Psi$. But we know almost nothing about type context Θ_1. However, we can lift this case to the one of $k = 2$ in that we derive $\Theta \vdash \Phi <: \Phi$ (or $\Theta \vdash \Psi <: \Psi$) using rule ($<:$ reflexivity).

Consider the case $k = 2$. Here we have proved subtypes $\Theta \vdash \Phi <: \Upsilon_1$ and $\Theta \vdash \Upsilon_1 <: \Psi$. Clearly, we can find type contexts Θ_Φ and Θ_Ψ such that

- $\Theta_\Phi \sqsubseteq \Theta$ and $\Theta_\Psi \sqsubseteq \Theta$ hold,
- $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$ are proved types, and
- Θ_Φ and Θ_Ψ are minimal with respect to $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$.

Moreover, $\Theta_{\Phi\Psi}$ shall be defined to fulfill $\Theta_{\Phi\Psi} \sqsubseteq \Theta$ and $|\Theta_{\Phi\Psi}| = |\Theta_\Phi| \cup |\Theta_\Psi|$. From consistency of Θ_Φ and Θ_Ψ (which we get due to Lemma 4.18 applied to $\Theta_\Phi \vdash \Phi$ and $\Theta_\Psi \vdash \Psi$) we immediately conclude consistency of $\Theta_{\Phi\Psi}$. By definition of function *sta*, it is clear that $\big((\varepsilon, \Phi), (\varepsilon, \Psi)\big) \in sta(\Phi) \times sta(\Psi)$. Hence, Lemma 4.46(iv) yields a type context $\widehat{\Theta}_{\Phi\Psi}$ with $\Theta_{\Phi\Psi} \sqsubseteq \widehat{\Theta}_{\Phi\Psi}$ such that $\widehat{\Theta}_{\Phi\Psi} \vdash \Phi <: \Psi$ is a proved subtype. Thus, there exists also a minimal type context $\Theta_{\min} \sqsubseteq \widehat{\Theta}_{\Phi\Psi}$ such that $\Theta_{\min} \vdash \Phi <: \Psi$ can be derived. But by Proposition 4.43, we then get $|\Theta_{\min}| = |\Theta_\Phi| \cup |\Theta_\Psi|$, which entails $\Theta_{\min} = \Theta_{\Phi\Psi}$. Hence, we have proved subtype $\Theta_{\Phi\Psi} \vdash \Phi <: \Psi$.

And since we already know $\Theta_{\Phi\Psi} \sqsubseteq \Theta$ and get consistency of Θ by Lemma 4.18 from proved type $\Theta \vdash \Phi$, for instance, we may apply (subtype weakening) to eventually obtain proved subtype $\Theta \vdash \Phi <: \Psi$.

In case of $k > 2$, we need to reconsider the proofs of propositions (iii) and (iv) of Lemma 4.46. Whenever we have been using proved subtypes $\Theta_{\Phi\Upsilon} \vdash \Phi <: \Upsilon$ and $\Theta_{\Upsilon\Psi} \vdash \Upsilon <: \Psi$ in the corresponding proofs, we only deduced the possible forms of one of the type terms Φ, Υ or Ψ from the shape of one of the other type terms or we have done the same for their subterms (either by inspection of the derivation rules for proved subtypes or by reference to the axioms for the subtype relation $<:$, using Proposition 4.16). However, this sort of deduction of forms can also be based on chains of proved subtypes

$$\Theta_1 \vdash \Upsilon_0 <: \Upsilon_1 \,, \quad \Theta_2 \vdash \Upsilon_1 <: \Upsilon_2 \quad , \ldots , \quad \Theta_k \vdash \Upsilon_{k-1} <: \Upsilon_k \,.$$

Hence, we may very well extend Lemma 4.46 (iii) and (iv) to this case of finite chains of proved subtypes. Then, however, our argument for case $k = 2$ will also be applicable for the case of any $k > 2$. □

One may ask whether this weak form of transitivity is sufficient. Recall we have introduced the notion of proved types because we wanted to distinguish type terms that we regard as well-formed. Hence, we do actually not need transitivity for proved subtypes which relate non-derivable types (at the lower and upper end of chains of proved subtypes).

This finally concludes our detailed investigation of transitivity of proved subtypes.

4.2.5 Towards type preservation

In this subsection we lay the necessary foundations (on the level of types) for a later proof of preservation of well-typedness. In [Pie02] Pierce describes the notion of preservation as follows:

> "If a well-typed term takes a step of evaluation, then the resulting term is also well typed."
> ([Pie02], page 95)

As Pierce points out in a footnote related to this sentence, in some type systems "evaluation preserves not only well-typedness but the exact types of terms." ([Pie02], footnote 3 on page 95). With the design of our type system, we also aim at such a strong notion of preservation.

How evaluation proceeds exactly for polymorphically typed *QLC* will be pinned down in section 4.5. Since we develop a polymorphic type system based on parametric polymorphism, there will be evaluation steps reducing type applications with the help of substitution of type variables. As a consequence, we need to establish a result (in a linear and a nonlinear version) showing that derivability of a type is preserved if we substitute all occurrences of a free type variable, say X, by a (well-formed) type term that respects the type bound of X. We may then remove the variable-type pair $X{<:}\Phi_X$ from the associated type context as long as we also perform the mentioned substitution in the remaining type context (right of $X{<:}\Phi_X$) itself.

But before we present the mentioned result, we first agree on the following notation: let $\Theta = X_1{<:}\Phi_1, \ldots, X_n{<:}\Phi_n$ be a type context. We write $\mathit{nftyv}(\Theta)$ to denote $\bigcup_{i=1}^{n} \mathit{nftyv}(\Phi_i)$. If Θ is empty, we write $\mathit{nftyv}(\emptyset)$, defined to be the empty set.

Similarly, by $\Theta[\Upsilon/X]$ we mean $X_1{<:}\Phi_1[\Upsilon/X], \ldots, X_n{<:}\Phi_n[\Upsilon/X]$.

We start with the linear case:

Lemma 4.48. *We can derive rule*

$$\frac{\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi \qquad \Theta \vdash \Upsilon \qquad \Theta \vdash \Upsilon <: \Phi_X \qquad \begin{array}{c} X \notin \mathit{nftyv}(\Theta') \\ X \notin \mathit{nftyv}(\Psi) \end{array}}{\Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X]}\ \textit{(linear type substitution)}\ . \qquad (\star)$$

Proof. We show this by nested induction on the length of type context Θ' (outer induction) and on the derivation of $\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi$ (inner induction).

Outer base case: Let Θ' be the empty context, denoted $\Theta' = \emptyset$.

Inner base cases: Suppose $\Theta, X{<:}\Phi_X \vdash \Psi$ has been derived using one of the rules (*Top* type), (*Unit* type) or (*Qbit* type), i.e. $\Psi \in \{!^n \mathit{Top}, !^n \mathit{Unit}, \mathit{Qbit} \mid n \geq 0\}$. Then $\Psi[\Upsilon/X] = \Psi$. Hence, $\Theta \vdash \Psi[\Upsilon/X]$ is obviously a proved type.

Suppose $\Theta, X{<:}\Phi_X \vdash Y$ has been derived using rule (linear type variable). We distinguish two cases:

Case $Y \neq X$: We then know Θ is of the form $\Xi, Y{<:}\Phi_Y, \Xi'$. But then $\Xi, Y{<:}\Phi_Y, \Xi' \vdash \underbrace{Y[\Upsilon/X]}_{=Y}$ is derivable by rule (linear type variable).

Case $Y = X$: From premise $\Theta \vdash \Upsilon$ in $(\star)$ we know $\Theta \vdash X[\Upsilon/X] = \Theta \vdash \Upsilon$ is a proved type.

Suppose $\Theta, X{<:}\Phi_X \vdash !^n Y$ has been derived using rule (nonlinear type variable). We distinguish two cases:

Case $Y \neq X$: We then know Θ is of the form $\Xi, Y{<:}!\Phi_Y, \Xi'$. But then $\Xi, Y{<:}!\Phi_Y, \Xi' \vdash \underbrace{(!^n Y)[\Upsilon/X]}_{= !^n Y}$ is derivable by (nonlinear type variable).

Case $Y = X$: In this case we know $n = 0$, because of premise $X \notin \mathit{nftyv}(!^n X)$ in $(\star)$. From premise $\Theta \vdash \Upsilon$ in $(\star)$ we conclude $\Theta \vdash (!^0 X)[\Upsilon/X] = \Theta \vdash \Upsilon$ is a proved type.

Inner induction cases: Assume $\Theta, X{<:}\Phi_X \vdash !^n(\Phi' \multimap \Psi')$ has been derived using rule (function type). By inner induction (where $X \notin \mathit{nftyv}(\Phi')$ and $X \notin \mathit{nftyv}(\Psi')$ are ensured by premise $X \notin \mathit{nftyv}(!^n(\Phi' \multimap \Psi'))$ in $(\star)$), it follows both $\Theta \vdash \Phi'[\Upsilon/X]$ and $\Theta \vdash \Psi'[\Upsilon/X]$ are derivable proved types. Using these as premises, we apply rule (function type) to obtain $\Theta \vdash !^n(\Phi'[\Upsilon/X] \multimap \Psi'[\Upsilon/X])$, which equals $\Theta \vdash (!^n(\Phi' \multimap \Psi'))[\Upsilon/X]$ by definition of substitution. Similar arguments cover rules (product type) and (sum type).

Consider the case where $\Theta, X{<:}\Phi_X \vdash \Psi$ has been derived by

$$\frac{\vdash \Xi, \Xi', X{<:}\Phi_X \qquad \Xi, Y{<:}\Phi_Y, \Xi', X{<:}\Phi_X \vdash \Psi' \qquad Y \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi', X{<:}\Phi_X \vdash (\forall Y{<:}\Phi_Y.\Psi')}\ \text{(linear-polymorphic type)}\ , \qquad (\star 2)$$

where we have $\Xi, \Xi' = \Theta$ and may assume $X \neq Y$ using α-equivalence (we will not explicitly mention this assumption again in the subsequent cases, but rather just implicitly use it). From premise $\vdash \Xi, \Xi', X{<:}\Phi_X$ we know Ξ, Ξ' is consistent. By inner induction (where $X \notin \mathit{nftyv}(\Psi')$ follows from premise $X \notin \mathit{nftyv}((\forall Y{<:}\Phi_Y.\Psi'))$ in $(\star)$), we may additionally conclude $\Xi, Y{<:}\Phi_Y, \Xi' \vdash \Psi'[\Upsilon/X]$ is a proved type. Furthermore, it is clear that $Y \notin \mathit{ftyv}(\Upsilon)$, because of premise $\Xi, \Xi' \vdash \Upsilon$ in $(\star)$, Lemma 4.20 and the fact that $\Xi, Y{<:}\Phi_Y, \Xi', X{<:}\Phi$ in the second premise of $(\star 2)$ is assumed to be a proper context, i.e. $Y \notin |\Xi, \Xi'|$. Hence, $Y \notin \mathit{nftyv}(\Upsilon)$ holds, as we get $\mathit{nftyv}(\Upsilon) \subseteq \mathit{ftyv}(\Upsilon)$ from Proposition 4.6. At the end of this chain of arguments, we see $Y \notin \mathit{nftyv}(\Psi'[\Upsilon/X])$, due to premise $Y \notin \mathit{nftyv}(\Psi')$ in $(\star 2)$ and the fact that substitution does not introduce fresh free type variables. Having these facts together, we may perform derivation

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, Y{<:}\Phi_Y, \Xi' \vdash \Psi'[\Upsilon/X] \qquad Y \notin \mathit{nftyv}(\Psi'[\Upsilon/X])}{\Xi, \Xi' \vdash (\forall Y{<:}\Phi_Y.\Psi'[\Upsilon/X])}\ \text{(linear-polymorphic type)}\ .$$

Moreover, consistency of $\Xi, Y{<:}\Phi_Y, \Xi'$ (due to Lemma 4.18 and the second premise of $(\star 2)$) implies $\Xi \vdash \Phi_Y$ and by Lemma 4.20 this leads to $\mathit{ftyv}(\Phi_Y) \subseteq |\Xi|$. On the other hand, we know $X \notin |\Xi|$, since $\Xi, Y{<:}\Phi_Y, \Xi', X{<:}\Phi_X$ is assumed to be a properly defined type context. Thus, it clearly holds $X \notin \mathit{ftyv}(\Phi_Y)$, and hence, it is evident that $\Phi_Y[\Upsilon/X] = \Phi_Y$. Finally, this entails

$$(\forall Y{<:}\Phi_Y.\Psi'[\Upsilon/X]) = (\forall Y{<:}\Phi_Y[\Upsilon/X].\Psi'[\Upsilon/X]) = (\forall Y{<:}\Phi_Y.\Psi')[\Upsilon/X]\ ,$$

and thus we know $\Xi, \Xi' \vdash (\forall Y{<:}\Phi_Y.\Psi')[\Upsilon/X] = \Theta \vdash \Psi[\Upsilon/X]$ is a proved type.

Now consider the case where $\Theta, X{<:}\Phi_X \vdash \Psi$ has been derived in the following way:

$$\frac{\vdash \Theta, X{<:}\Phi_X \qquad \Theta, X{<:}\Phi_X, Y{<:}\Phi_Y \vdash \Psi' \qquad Y \notin \mathit{nftyv}(\Psi')}{\Theta, X{<:}\Phi_X \vdash (\forall Y{<:}\Phi_Y.\Psi')}\ \text{(linear-polymorphic type)}\ . \qquad (\star 3)$$

From the second premise, Lemma 4.18 and the definition of consistency we get proved type $\Theta, X{<:}\Phi_X \vdash \Phi_Y$, which additionally entails (also by Lemma 4.18) consistency of $\Theta, X{<:}\Phi_X$. By inner induction (where $X \notin \mathit{nftyv}(\emptyset)$ clearly holds and $X \notin \mathit{nftyv}(\Phi_Y)$ is ensured by premise $X \notin \mathit{nftyv}((\forall Y{<:}\Phi_Y.\Psi')) = \mathit{nftyv}(!^k \Phi_Y) \cup (\mathit{nftyv}(\Psi') \setminus \{Y\})$ in $(\star)$ with $k = 0$, due to premise $Y \notin \mathit{nftyv}(\Psi')$ in $(\star 3)$) we then come to $\Theta \vdash \Phi_Y[\Upsilon/X]$ which we weaken to $\Theta, X{<:}\Phi_X \vdash \Phi_Y[\Upsilon/X]$. This is the right point to apply rule (type bound replacement) in derivation

$$\frac{\Theta, X{<:}\Phi_X \vdash !^m \Phi'_Y[\Upsilon/X] \qquad \Phi'_Y \text{ linear} \qquad \Theta, X{<:}\Phi_X, Y{<:}!^m \Phi'_Y \vdash \Psi' \qquad m > 0 \Rightarrow m' > 0}{\Theta, X{<:}\Phi_X, Y{<:}!^m \Phi'_Y[\Upsilon/X] \vdash \Psi'}\ \text{(type bound replacement)}\ ,$$

where we find $\Phi_Y = !^m\Phi'_Y$ for some linear type term Φ'_Y and where $m' \geq m \geq 0$ is the number of leading exponentials in $!^m\Phi'_Y[\Upsilon/X]$. But $\Theta \vdash \Phi_Y[\Upsilon/X]$ also entails consistency of $\Theta, Y{<:}\Phi_Y[\Upsilon/X]$. Thus, we can perform a weakening step

$$\frac{\vdash \Theta, Y{<:}\Phi_Y[\Upsilon/X] \qquad \Theta \vdash \Phi_X \qquad \Theta \sqsubseteq \Theta, Y{<:}\Phi_Y[\Upsilon/X]}{\Theta, Y{<:}\Phi_Y[\Upsilon/X] \vdash \Phi_X}\ \text{(type weakening)}$$

and thus get consistency of $\Theta, Y <: \Phi_Y[\Upsilon/X], X <: \Phi_X$. Those two results together enable context permutation

$$\frac{\vdash \Theta, Y{<:}\Phi_Y[\Upsilon/X], X{<:}\Phi_X \qquad \Theta, X{<:}\Phi_X, Y{<:}\Phi_Y[\Upsilon/X] \vdash \Psi'}{\Theta, Y{<:}\Phi_Y[\Upsilon/X], X{<:}\Phi_X \vdash \Psi'}\ \text{(type permutation)}\ .$$

Now we use inner induction again (where $X \notin \mathit{nftyv}(Y{<:}\Phi_Y)$ and $X \notin \mathit{nftyv}(\Psi')$ hold due to premise $X \notin \mathit{nftyv}((\forall Y{<:}\Phi_Y.\Psi'))$ in $(\star)$) leading to $\Theta, Y{<:}\Phi_Y[\Upsilon/X] \vdash \Psi'[\Upsilon/X]$. Finally, we apply (linear-polymorphic type) in derivation

$$\frac{\vdash \Theta \qquad \Theta, Y{<:}\Phi_Y[\Upsilon/X] \vdash \Psi'[\Upsilon/X] \qquad Y \notin \mathit{nftyv}(\Psi'[\Upsilon/X])}{\Theta \vdash \underbrace{(\forall Y{<:}\Phi_Y[\Upsilon/X].\Psi'[\Upsilon/X])}_{= (\forall Y{<:}\Phi_Y.\Psi')[\Upsilon/X]}}\ \text{(linear-polymorphic type)}\ ,$$

where we conclude $Y \notin \mathit{nftyv}(\Psi'[\Upsilon/X])$ from $Y \notin \mathit{nftyv}(\Psi')$ (the third premise in $(\star 3)$) and $Y \notin \mathit{nftyv}(\Upsilon)$ (which we will show immediately) and the obvious fact that substitution does not introduce any fresh free type variables. $Y \notin \mathit{nftyv}(\Upsilon)$ follows from the assumption that $\Theta, X{<:}\Phi_X, Y{<:}\Phi_Y$ in the second premise of $(\star 3)$ is a properly defined type context which entails $Y \notin |\Theta|$. Furthermore, premise $\Theta \vdash \Upsilon$ in $(\star)$ plus Lemma 4.20 and Proposition 4.6 imply $\mathit{nftyv}(\Upsilon) \subseteq \mathit{ftyv}(\Upsilon) \subseteq |\Theta|$.

Suppose $\Theta, X{<:}\Phi_X \vdash \Psi$ has been derived by

$$\frac{\Xi, \Xi', X{<:}\Phi_X \vdash \Phi_Y <: !\mathit{Top} \qquad \Xi, Y{<:}\Phi_Y, \Xi', X{<:}\Phi_X \vdash !^n\Psi'}{\Xi, \Xi', X{<:}\Phi_X \vdash !^n(\forall Y{<:}\Phi_Y.\Psi')}\ \text{(nonlinear-polymorphic type)}$$

with $\Xi, \Xi' = \Theta$. From premise $\Xi, \Xi', X{<:}\Phi_X \vdash \Phi_Y <: !\mathit{Top}$ and Lemma 4.19, we know Ξ, Ξ' is consistent. It then follows, by inner induction (where $X \notin \mathit{nftyv}(!^n\Psi)$ holds due to premise $X \notin \mathit{nftyv}(!^n(\forall Y{<:}\Phi_Y.\Psi'))$ in $(\star)$), that $\Xi, Y{<:}\Phi_Y, \Xi' \vdash (!^n\Psi')[\Upsilon/X]$ is a proved type. When we apply Lemma 4.18 to this, we get proved type $\Xi \vdash \Phi_Y$, to which we apply rules (type weakening), (Top supertype) and (! right) (with the help of Corollary 4.17) to finally obtain proved subtype $\Xi, \Xi' \vdash \Phi_Y <: !\mathit{Top}$. Having this, we perform derivation

$$\frac{\Xi, \Xi' \vdash \Phi_Y <: !\mathit{Top} \qquad \Xi, Y{<:}\Phi_Y, \Xi' \vdash !^n\Psi'[\Upsilon/X]}{\Xi, \Xi' \vdash !^n(\forall Y{<:}\Phi_Y.\Psi'[\Upsilon/X])}\ \text{(nonlinear-polymorphic type)}\ .$$

As in the analogous case of rule (linear-polymorphic type), we eventually come up with $X \notin \mathit{ftyv}(\Phi_Y)$, and hence, $\Phi_Y[\Upsilon/X] = \Phi_Y$. Arguing further along the same lines, we conclude $\Xi, \Xi' \vdash (!^n(\forall Y{<:}\Phi_Y.\Psi'))[\Upsilon/X] = \Theta \vdash \Psi[\Upsilon/X]$ is a proved type.

Next, assume $\Theta, X{<:}\Phi_X \vdash \Psi$ has been derived by

$$\frac{\Theta, X{<:}\Phi_X \vdash \Phi_Y <: !\mathit{Top} \qquad \Theta, X{<:}\Phi_X, Y{<:}\Phi_Y \vdash !^n\Psi'}{\Theta, X{<:}\Phi_X \vdash !^n(\forall Y{<:}\Phi_Y.\Psi')}\ \text{(nonlinear-polymorphic type)}\ ,$$

Then we proceed in the same way as in the analogous case of rule (linear-polymorphic type) to obtain consistency of $\Theta, X{<:}\Phi_X, Y{<:}\Phi_Y[\Upsilon/X]$ and also of $\Theta, Y{<:}\Phi_Y[\Upsilon/X], X{<:}\Phi_X$ from the second premise. Application of rule (type bound replacement) using the former type context yields derivation

$$\frac{\begin{array}{cc}\Theta, X{<:}\Phi_X \vdash !^m\Phi'_Y[\Upsilon/X] & \Phi'_Y \text{ linear} \\ \Theta, X{<:}\Phi_X, Y{<:}!^m\Phi'_Y \vdash !^n\Psi' & m > 0 \Rightarrow m' > 0\end{array}}{\Theta, X{<:}\Phi_X, Y{<:}!^m\Phi'_Y[\Upsilon/X] \vdash !^n\Psi'}\ \text{(type bound replacement)}\ ,$$

where we analogously have $\Phi_Y = !^m\Phi'_Y$ and with $m' \geq m \geq 0$ being the number of leading exponentials in $!^m\Phi'_Y[\Upsilon/X]$. In this way we obtain the needed premises to perform context permutation

$$\frac{\vdash \Theta, Y{<:}\Phi_Y[\Upsilon/X], X{<:}\Phi_X \qquad \Theta, X{<:}\Phi_X, Y{<:}\Phi_Y[\Upsilon/X] \vdash !^n\Psi'}{\Theta, Y{<:}\Phi_Y[\Upsilon/X], X{<:}\Phi_X \vdash !^n\Psi'}\ \text{(type permutation)}\ .$$

Now we use inner induction again (for which $X \notin \mathit{nftyv}(!^n\Psi')$ is ensured by premise $X \notin \mathit{nftyv}(!^n(\forall Y{<:}\Phi_Y.\Psi'))$ in $(\star)$, and $X \notin \mathit{nftyv}(\emptyset)$ obviously holds) to obtain proved type $\Theta, Y{<:}\Phi_Y[\Upsilon/X] \vdash (!^n\Psi')[\Upsilon/X]$. Finally, we apply (nonlinear-polymorphic type) in derivation

$$\frac{\Theta \vdash \Phi_Y[\Upsilon/X] <: !\mathit{Top} \qquad \Theta, Y{<:}\Phi_Y[\Upsilon/X] \vdash !^n\Psi'[\Upsilon/X]}{\Theta \vdash \underbrace{!^n(\forall Y{<:}\Phi_Y[\Upsilon/X].\Psi'[\Upsilon/X])}_{= (!^n(\forall Y{<:}\Phi_Y.\Psi'))[\Upsilon/X]}}\ \text{(nonlinear-polymorphic type)}\ ,$$

where $\Theta \vdash \Phi_Y[\Upsilon/X] <: !\mathit{Top}$ can easily be derived using $\Theta \vdash \Phi_Y[\Upsilon/X]$ as a starting point to apply rules (*Top* supertype) and (! right), supported by Corollary 4.17.

This finishes the base case of our outer induction.

Outer induction case: Let $\Theta' = \Theta'', Y{<:}\Phi_Y$ be a non-empty type context with $X \neq Y$.

Inner base cases: Suppose $\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi$ has been derived using one of the rules (*Top* type), (*Unit* type) or (*Qbit* type). Then $\Psi[\Upsilon/X] = \Psi$, and we can easily derive $\Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X]$ using the same rule, where consistency of $\Theta, \Theta'[\Upsilon/X] = \Theta, \Theta''[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]$ is ensured by outer induction (using premise $X \notin \mathit{nftyv}(\Theta'', Y{<:}\Phi_Y)$ in $(\star)$ to conclude $X \notin \mathit{nftyv}(\Theta'')$ and $X \notin \mathit{nftyv}(\Phi_Y)$) applied to $\Theta, X{<:}\Phi_X, \Theta'' \vdash \Phi_Y$ (which stems from consistency of $\Theta, X{<:}\Phi_X, \Theta'$).

Suppose $\Theta, X{<:}\Phi_X, \Theta' \vdash Z$ has been derived using rule (linear type variable). We distinguish two cases:

Case $Z \neq X$ where either Θ or Θ' is of the form $\Xi, Z{<:}\Phi_Z, \Xi'$ for two type contexts Ξ and Ξ'. Because of $Z[\Upsilon/X] = Z$, we easily derive $\Theta, \Theta'[\Upsilon/X] \vdash Z[\Upsilon/X]$ using (linear type variable), where we conclude consistency of $\Theta, \Theta'[\Upsilon/X]$ using outer induction, as in the above case.

Case $Z = X$. From premise $\Theta \vdash \Upsilon$ in $(\star)$ we immediately get $\Theta \vdash X[\Upsilon/X] = \Theta \vdash \Upsilon$ as a proved type. Analogously to the two preceding cases, we may conclude consistency of $\Theta, \Theta'[\Upsilon/X]$ using outer induction. Application of (type weakening) then yields proved type $\Theta, \Theta'[\Upsilon/X] \vdash \Upsilon$.

Suppose $\Theta, X <: \Phi_X, \Theta' \vdash !^n Z$ has been derived using rule (nonlinear type variable). We distinguish two cases:

Case $Z \neq X$ where either Θ or Θ' is of the form $\Xi, Z{<:}!\Phi_Z, \Xi'$ for two type contexts Ξ and Ξ'. Because of $(!^n Z)[\Upsilon/X] = !^n Z$, we easily derive $\Theta, \Theta'[\Upsilon/X] \vdash (!^n Z)[\Upsilon/X]$ using (nonlinear type variable), where we conclude consistency of $\Theta, \Theta'[\Upsilon/X]$ using outer induction, as in the above case.

Case $Z = X$. In this case we know $n = 0$, because of premise $X \notin \mathit{nftyv}(!^n X)$ in $(\star)$. Now we are in a similar situation as in the respective case of rule (linear type variable), and may thus reuse the same arguments to come to proved type $\Theta, \Theta'[\Upsilon/X] \vdash \Upsilon$.

Inner induction cases: Assume $\Theta, X{<:}\Phi_X, \Theta' \vdash !^n(\Phi' \multimap \Psi')$ has been derived using rule (function type). By inner induction (for which $X \notin \mathit{nftyv}(\Phi')$ and $X \notin \mathit{nftyv}(\Psi')$ hold due to premise $X \notin \mathit{nftyv}(!^n(\Phi' \multimap \Psi'))$ in $(\star)$), it follows that both $\Theta, \Theta'[\Upsilon/X] \vdash \Phi'[\Upsilon/X]$ and $\Theta, \Theta'[\Upsilon/X] \vdash \Psi'[\Upsilon/X]$ are derivable proved types. Using these as premises to apply rule (function type), we get $\Theta, \Theta'[\Upsilon/X] \vdash !^n(\Phi'[\Upsilon/X] \multimap \Psi'[\Upsilon/X])$, which equals $\Theta, \Theta'[\Upsilon/X] \vdash (!^n(\Phi' \multimap \Psi'))[\Upsilon/X]$. Similar arguments cover rules (product type) and (sum type).

In the subsequent cases we implicitly assume $X \neq Z$ using α-equivalence.

Consider the case where $\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi$ has been derived by

$$\frac{\vdash \Xi, \Xi', X{<:}\Phi_X, \Theta' \qquad \Xi, Z{<:}\Phi_Z, \Xi', X{<:}\Phi_X, \Theta' \vdash \Psi' \qquad Z \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi', X{<:}\Phi_X, \Theta' \vdash (\forall Z{<:}\Phi_Z.\Psi')}\ \text{(linear-polymorphic type)}\ . \qquad (\star 4)$$

(Remember Θ' is of the form $\Theta'', Y{<:}\Phi_Y$ in this case.) From the first premise of this derivation, Lemma 4.18 and the definition of consistency we know $\Xi, \Xi', X{<:}\Phi_X, \Theta'' \vdash \Phi_Y$ is a proved type. Outer induction tells us we may apply rule (linear type substitution) as follows:

$$\frac{\Xi, \Xi', X{<:}\Phi_X, \Theta'' \vdash \Phi_Y \qquad \Xi, \Xi' \vdash \Upsilon \qquad \Xi, \Xi' \vdash \Upsilon{<:}\Phi_X \qquad X \notin \mathit{nftyv}(\Theta'') \qquad X \notin \mathit{nftyv}(\Phi_Y)}{\Xi, \Xi', \Theta''[\Upsilon/X] \vdash \Phi_Y[\Upsilon/X]}\ \text{(linear type substitution)}\ ,$$

where we know $X \notin \mathit{nftyv}(\Theta'')$ and $X \notin \mathit{nftyv}(\Phi_Y)$ from premise $X \notin \mathit{nftyv}(\Theta'', Y{<:}\Phi_Y)$ in $(\star)$. Clearly, this leads to consistency of $\Xi, \Xi', \Theta''[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] = \Xi, \Xi', \Theta'[\Upsilon/X]$. By inner induction we may additionally conclude $\Xi, Z{<:}\Phi_Z, \Xi', \Theta'[\Upsilon/X] \vdash \Psi'[\Upsilon/X]$ is a proved type.

Furthermore, it is clear that $Z \notin \mathit{ftyv}(\Upsilon)$, because of premise $\Xi, \Xi' \vdash \Upsilon$ in $(\star)$, Lemma 4.20 and the fact that $\Xi, Z{<:}\Phi_Z, \Xi', X{<:}\Phi_X, \Theta'$ in the second premise of $(\star 4)$ is assumed to be a proper type context, i.e. $Z \notin |\Xi, \Xi'|$. Hence, $Z \notin \mathit{nftyv}(\Upsilon)$ holds since we have $\mathit{nftyv}(\Upsilon) \subseteq \mathit{ftyv}(\Upsilon)$ due to Proposition 4.6. Thus, we obtain $Z \notin \mathit{nftyv}(\Psi'[\Upsilon/X])$, since $Z \notin \mathit{nftyv}(\Psi')$ is the third premise of $(\star 4)$ and due to the fact that substitution does not introduce any fresh free type variables. Having these facts together, we may perform

$$\frac{\vdash \Xi, \Xi', \Theta'[\Upsilon/X] \qquad \Xi, Z{<:}\Phi_Z, \Xi', \Theta'[\Upsilon/X] \vdash \Psi'[\Upsilon/X] \qquad Z \notin \mathit{nftyv}(\Psi'[\Upsilon/X])}{\Xi, \Xi', \Theta'[\Upsilon/X] \vdash (\forall Z{<:}\Phi_Z.\Psi'[\Upsilon/X])}\ \text{(linear-polymorphic type)}\ .$$

From consistency of $\Xi, Z{<:}\Phi_Z, \Xi', X{<:}\Phi_X$ we know two things. Firstly, $\Xi \vdash \Phi_Z$ holds and entails (according to Lemma 4.20) $\mathit{ftyv}(\Phi_Z) \subseteq |\Xi|$. Secondly, we know $X \notin |\Xi|$, according to the definition of type contexts. Hence, we know $X \notin \mathit{ftyv}(\Phi_Z) \subseteq |\Xi|$. This obviously entails $\Phi_Z[\Upsilon/X] = \Phi_Z$ and thus also $(\forall Z{<:}\Phi_Z.\Psi'[\Upsilon/X]) = (\forall Z{<:}\Phi_Z[\Upsilon/X].\Psi'[\Upsilon/X]) = (\forall Z{<:}\Phi_Z.\Psi')[\Upsilon/X]$, eventually leading to the result that $\Xi, \Xi', \Theta'[\Upsilon/X] \vdash (\forall Z{<:}\Phi_Z.\Psi')[\Upsilon/X]$ is a proved type.

Now that we have studied the previous case in full detail, we proceed in a less detailed way in the remaining steps since the way of reasoning is very similar.

Consider the case where $\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi$ has been derived by

$$\frac{\vdash \Theta, X{<:}\Phi_X, \Xi, \Xi', Y{<:}\Phi_Y \qquad \Theta, X{<:}\Phi_X, \Xi, Z{<:}\Phi_Z, \Xi', Y{<:}\Phi_Y \vdash \Psi' \qquad Z \notin \mathit{nftyv}(\Psi')}{\Theta, X{<:}\Phi_X, \Xi, \Xi', Y{<:}\Phi_Y \vdash (\forall Z{<:}\Phi_Z.\Psi')}\ \text{(linear-polymorphic type)}\ . \qquad (\star 5)$$

Inner induction yields

$$\frac{\Theta, X{<:}\Phi_X, \Xi, Z{<:}\Phi_Z, \Xi', Y{<:}\Phi_Y \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Xi, Z{<:}\Phi_Z, \Xi', Y{<:}\Phi_Y) \qquad \Theta \vdash \Upsilon \qquad \Theta \vdash \Upsilon <: \Phi_X \qquad X \notin \mathit{nftyv}(\Psi')}{\Theta, \Xi[\Upsilon/X], Z{<:}\Phi_Z[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash \Psi'[\Upsilon/X]}\ \text{(linear type substitution)}\ .$$

Outer induction yields

$$\frac{\Theta, X{<:}\Phi_X, \Xi, \Xi' \vdash \Phi_Y \qquad \Theta \vdash \Upsilon \qquad \Theta \vdash \Upsilon{<:}\Phi \qquad X \notin \mathit{nftyv}(\Xi, \Xi') \qquad X \notin \mathit{nftyv}(\Phi_Y)}{\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X] \vdash \Phi_Y[\Upsilon/X]}\ \text{(linear type substitution)}\ ,$$

which entails consistency of $\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]$. $Z \notin \mathit{nftyv}(\Psi'[\Upsilon/X])$ follows from premise $Z \notin \mathit{nftyv}(\Psi')$ in rule (linear-polymorphic type), from premise $\Theta \vdash \Upsilon$ in $(\star)$ which entails $\mathit{nftyv}(\Upsilon) \subseteq \mathit{ftyv}(\Upsilon) \subseteq |\Theta|$ and the fact that $\Theta, X{<:}\Phi_X, \Xi, Z{<:}\Phi_Z, \Xi', Y{<:}\Phi_Y$ is a proper type context (and thus $Z \notin |\Theta|$). Furthermore, substitution does not introduce any fresh free type variables. Now we have all necessary pieces together to perform derivation

$$\frac{\vdash \Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \qquad Z \notin \mathit{nftyv}(\Psi'[\Upsilon/X]) \quad \Theta, \Xi[\Upsilon/X], Z{<:}\Phi_Z[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash \Psi'[\Upsilon/X]}{\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash (\forall Z{<:}\Phi_Z[\Upsilon/X].\Psi'[\Upsilon/X])}\ \text{(linear-polymorphic type)}$$

to obtain proved type

$$\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash (\forall Z{<:}\Phi_Z.\Psi')[\Upsilon/X] \ .$$

Consider the case where $\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi$ has been derived by

$$\frac{\Xi, \Xi', X{<:}\Phi_X, \Theta'', Y{<:}\Phi_Y \vdash \Phi_Z <: !\mathit{Top} \quad \Xi, Z{<:}\Phi_Z, \Xi', X{<:}\Phi_X, \Theta'', Y{<:}\Phi_Y \vdash !^n\Psi'}{\Xi, \Xi', X{<:}\Phi_X, \Theta'', Y{<:}\Phi_Y \vdash !^n(\forall Z{<:}\Phi_Z.\Psi')}\ \text{(nonlinear-polymorphic type)} \ .$$

Inner induction then yields

$$\Xi, Z{<:}\Phi_Z, \Xi', \underbrace{\Theta''[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]}_{=\ \Theta'[\Upsilon/X]} \vdash (!^n\Psi')[\Upsilon/X] \ .$$

Outer induction yields

$$\Xi, \Xi', \Theta''[\Upsilon/X] \vdash \Phi_Y[\Upsilon/X] \ .$$

which implies consistency of $\Xi, \Xi', \underbrace{\Theta''[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]}_{=\ \Theta'[\Upsilon/X]}$.

From the second premise and Lemma 4.18 we get $\Xi \vdash \Phi_Z$. Furthermore, the first premise and Corollary 4.17 tell us Φ_Z is of the form $!\Phi'_Z$, which enables derivation

$$\frac{\dfrac{\Xi \vdash \Phi_Z}{\Xi \vdash \Phi_Z <: \mathit{Top}}\ (\mathit{Top}\ \text{supertype})}{\Xi \vdash \Phi_Z <: !\mathit{Top}}\ (!\ \text{right}) \ .$$

Hence, we can apply (subtype weakening) as follows

$$\frac{\vdash \Xi, \Xi', \Theta'[\Upsilon/X] \qquad \Xi \vdash \Phi_Z <: !\mathit{Top} \qquad \Xi \sqsubseteq \Xi, \Xi', \Theta'[\Upsilon/X]}{\Xi, \Xi', \Theta'[\Upsilon/X] \vdash \Phi_Z <: !\mathit{Top}}\ \text{(subtype weakening)} \ .$$

The last result together with the proved type we obtained by inner induction now yield the necessary premises to perform derivation

$$\frac{\Xi, \Xi', \Theta'[\Upsilon/X] \vdash \Phi_Z <: !\mathit{Top} \quad \Xi, Z{<:}\Phi_Z, \Xi', \Theta'[\Upsilon/X] \vdash !^n\Psi'[\Upsilon/X]}{\Xi, \Xi', \Theta'[\Upsilon/X] \vdash !^n(\forall Z{<:}\Phi_Z.\Psi'[\Upsilon/X])}\ \text{(nonlinear-polymorphic type)} \ .$$

As in the respective case of rule (linear-polymorphic) type in the outer base case, we use Lemmas 4.18 and 4.20 to deduce $X \notin |\Xi|$ and $\mathit{ftyv}(\Phi_Z) \subseteq |\Xi|$, which entails $\Phi_Z[\Upsilon/X] = \Phi_Z$. Consequently, we already reached our goal, since

$$!^n(\forall Z{<:}\Phi_Z.\Psi'[\Upsilon/X]) = \ !^n(\forall Z{<:}\Phi_Z[\Upsilon/X].\Psi'[\Upsilon/X]) = (!^n(\forall Z{<:}\Phi_Z.\Psi'))[\Upsilon/X]$$

and hence, we have already derived proved type

$$\Xi, \Xi', \Theta'[\Upsilon/X] \vdash (!^n(\forall Z{<:}\Phi_Z.\Psi'))[\Upsilon/X] = \Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X] \ .$$

Consider the case where $\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi$ has been derived by

$$\frac{\Theta, X{<:}\Phi_X, \Xi, \Xi', Y{<:}\Phi_Y \vdash \Phi_Z <: !\mathit{Top} \qquad \Theta, X{<:}\Phi_X, \Xi, Z{<:}\Phi_Z, \Xi', Y{<:}\Phi_Y \vdash !^n\Psi'}{\Theta, X{<:}\Phi_X, \Xi, \Xi', Y{<:}\Phi_Y \vdash !^n(\forall Z{<:}\Phi_Z.\Psi')}\ \text{(nonlinear-polymorphic type)}\ .$$

Inner induction yields

$$\Theta, \Xi[\Upsilon/X], Z{<:}\Phi_Z[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash (!^n\Psi')[\Upsilon/X].$$

Outer induction yields

$$\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X] \vdash \Phi_Y[\Upsilon/X]\ ,$$

which entails consistency of $\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]$. From the result of inner induction and Lemma 4.18 we know $\Theta, \Xi[\Upsilon/X] \vdash \Phi_Z[\Upsilon/X]$. Using the consistency result we just obtained, we can weaken this proved type to $\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash \Phi_Z[\Upsilon/X]$. Then we apply rules (*Top* supertype) and (! right), supported by Corollary 4.17, to obtain proved subtype

$$\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash \Phi_Z[\Upsilon/X] <: !\mathit{Top}\ .$$

Now we have all necessary pieces together to perform derivation

$$\frac{\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash \Phi_Z[\Upsilon/X] <: !\mathit{Top} \qquad \Theta, \Xi[\Upsilon/X], Z{<:}\Phi_Z[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash !^n\Psi'[\Upsilon/X]}{\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash !^n(\forall Z{<:}\Phi_Z[\Upsilon/X].\Psi'[\Upsilon/X])}\ \text{(nonlinear-polymorphic type)}$$

to finally arrive at proved type

$$\Theta, \Xi[\Upsilon/X], \Xi'[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X] \vdash (!^n(\forall Z{<:}\Phi_Z.\Psi'))[\Upsilon/X] = \Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X]\ ,$$

which is exactly what we intended to derive.

□

Next we derive the rule that is complementary to (linear type substitution). Luckily, we can reuse the majority of the previous proof.

Lemma 4.49. *We can derive rule*

$$\frac{\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi \qquad \Theta \vdash !\Upsilon \quad \Theta \vdash \Upsilon <: \Phi_X}{\Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X]}\ \textit{(nonlinear type substitution)}\ . \qquad (\star)$$

Proof. When we compare rules (linear type substitution) and (nonlinear type substitution), we notice the differences lie solely in the premises. And since these differences are not too big, we can reuse the majority of the arguments applied in the proof for Lemma 4.48 for the proof of Lemma 4.49 (possibly in a slightly modified way). Hence, we here only treat the cases where the line of argument needs to be adjusted in a nontrivial way.

At first, by Lemma 4.31, we can see whenever we can derive a proved type $\Theta \vdash !\Upsilon$, then we can also derive $\Theta \vdash \Upsilon$ (by application of rule (! elimination), in other words). That means, we can assume $\Theta \vdash \Upsilon$ to be an implicit premise of rule (nonlinear type substitution). And obviously, we then do not have to modify any arguments from the proof of Lemma 4.48 that use this premise.

Furthermore, while the linear rule requires the substituted type variable not to appear as nonlinear free type variable in type context Θ' and in type term Ψ, the nonlinear substitution rule does not pose any such requirements. These requirements are of crucial importance in two cases which we treat in detail below. However, they are also needed for all steps where we applied (inner or outer) induction in the proof of Lemma 4.48. But since rule (nonlinear type substitution) does not pose such requirements, we do of course not have to ensure them when we want to use induction based on this rule. Thus, we are relieved of another duty in the proof of Lemma 4.49.

Let us treat the remaining (closely connected) cases then:

Outer base case, inner base case:
Suppose $\Theta, X{<:}\Phi_X \vdash Y$ has been derived using rule (linear type variable). We distinguish two cases:

Case $Y \neq X$: Analogous to the proof of Lemma 4.48.

Case $Y = X$: Rule (! elimination) applied to the second premise in $(\star)$ yields proved type $\Theta \vdash \Upsilon = \Theta \vdash X[\Upsilon/X]$.

Suppose $\Theta, X{<:}\Phi_X \vdash !^n Y$ has been derived using rule (nonlinear type variable). We distinguish two cases:

Case $Y \neq X$: Analogous to the proof of Lemma 4.48.

Case $Y = X$: By Lemma 4.31 we know that we can derive $\Theta \vdash !^{n'}\Upsilon$ for any $n' \geq 0$, whenever we can derive $\Theta \vdash !\Upsilon$. Hence, the second premise in $(\star)$ together with Lemma 4.31 entails $\Theta \vdash (!^n X)[\Upsilon/X] = \Theta \vdash !^n\Upsilon$ is a proved type.

Outer induction case, inner base case:
Suppose $\Theta, X{<:}\Phi_X, \Theta' \vdash Z$ has been derived using rule (linear type variable). We distinguish two cases:

Case $Z \neq X$: Analogous to the proof of Lemma 4.48.

Case $Z = X$: By rule (! elimination) applied to premise $\Theta \vdash !\Upsilon$ in $(\star)$, we immediately get $\Theta \vdash \Upsilon = \Theta \vdash X[\Upsilon/X]$ as a proved type. Consistency of $\Theta, \Theta'[\Upsilon/X] = \Theta, \Theta''[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]$ is ensured by outer induction applied to $\Theta, X{<:}\Phi_X, \Theta'' \vdash \Phi_Y$ (which stems from consistency of $\Theta, X{<:}\Phi_X, \Theta'', Y{<:}\Phi_Y$, and we got this due to Lemma 4.18 applied to the first premise of $(\star)$). Application of (type weakening) then yields proved type $\Theta, \Theta'[\Upsilon/X] \vdash \Upsilon$.

Suppose $\Theta, X{<:}\Phi_X, \Theta' \vdash !^n Z$ has been derived using rule (nonlinear type variable). We distinguish two cases:

Case $Z \neq X$: Analogous to the proof of Lemma 4.48.

Case $Z = X$: From premise $\Theta \vdash !\Upsilon$ in $(\star)$ and Lemma 4.31, we immediately get $\Theta \vdash !^n\Upsilon = \Theta \vdash !^n X[\Upsilon/X]$ as a proved type. Analogous to the previous case we can derive $\Theta, \Theta'[\Upsilon/X] \vdash !^n\Upsilon$ by weakening of proved type $\Theta \vdash !^n X[\Upsilon/X]$ (taking the intermediate step of ensuring consistency of $\Theta, \Theta'[\Upsilon/X] = \Theta, \Theta''[\Upsilon/X], Y{<:}\Phi_Y[\Upsilon/X]$ by outer induction).

□

Let us again take a short informal look at rules (linear type substitution) and (nonlinear type substitution). Both of them pose some restrictions either on the type variable X that is to be substituted in type term Ψ or on the type term Υ that is to be put in place of X in Ψ (besides the enforcement of X's type bound). While in the linear case X is not allowed to appear as nonlinear free type variable in type term Ψ (and its type context) and Υ is not subject to any restriction (other than $\Upsilon <: \Phi_X$), the picture changes to the opposite in the nonlinear case, where we do not care about how X occurs in Ψ, but rather restrict Υ to be a type term that can possibly be derived as a duplicable type. Put this way, we get a feeling that the two substitution rules with their respective restrictions complement each other perfectly. We have also seen and used this fact in the proof of Lemma 4.49, which needed to be adjusted in a few places only compared to the proof of Lemma 4.48. And in all of these places we were able to replace the restrictions used in one proof by the complementary restriction of the other rule, thus yielding the necessary conclusions.

To conclude this section, we establish a result which shows (at least at the level of type terms) that type application, as we will incorporate it to the operational semantics in section 4.5, preserves derivability of proved types. And in the respective proof we will again see the complementary nature of rules (linear type substitution) and its nonlinear counterpart. However, this time we will additionally notice how this fits together with the (perhaps less obvious) complementary nature of derivation rules (linear-polymorphic type) and (nonlinear-polymorphic type).

Theorem 4.50. *Given proved types* $\Theta \vdash !^n(\forall X{<:}\Phi_X.\Psi)$ *and* $\Theta \vdash \Upsilon$ *and proved subtype* $\Theta \vdash \Upsilon <: \Phi_X$*, we can derive proved type* $\Theta \vdash !^n\Psi[\Upsilon/X]$*.*

In other words, we can derive rule

$$\frac{\Theta \vdash !^n(\forall X{<:}\Phi_X.\Psi) \qquad \begin{array}{c}\Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array}}{\Theta \vdash !^n\Psi[\Upsilon/X]}\ \textit{(bounded type substitution)}\ .$$

Proof. We need to distinguish a linear and a nonlinear case:

Suppose $\Theta \vdash !^n(\forall X{<:}\Phi_X.\Psi)$ has been derived by

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi \qquad X \notin \mathit{nftyv}(\Psi)}{\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi)}\ \text{(linear-polymorphic type)}\ ,$$

where we immediately know $n = 0$. We thus write type context Θ as Ξ, Ξ', and know by Lemma 4.18 that $\Xi, X{<:}\Phi_X, \Xi'$ and hence also $\Xi, X{<:}\Phi_X$ are consistent. Type weakening and type permutation then yield

$$\frac{\vdash \Xi, \Xi' \qquad \Xi \vdash \Phi_X \qquad \Xi \sqsubseteq \Xi, \Xi'}{\Xi, \Xi' \vdash \Phi_X}\ \text{(type weakening)}$$

(i.e. $\Xi, \Xi', X{<:}\Phi_X$ is consistent) and

$$\frac{\vdash \Xi, \Xi', X{<:}\Phi_X \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi}{\underbrace{\Xi, \Xi', X{<:}\Phi_X \vdash \Psi}_{= \Theta, X{<:}\Phi_X \vdash \Psi}}\ \text{(type permutation).}$$

Now we are in the right position to use (linear type substitution) to perform

$$\frac{\Theta, X{<:}\Phi_X \vdash \Psi \qquad \begin{array}{c}\Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array} \qquad \begin{array}{c}X \notin \mathit{nftyv}(\emptyset) \\ X \notin \mathit{nftyv}(\Psi)\end{array}}{\Theta \vdash \Psi[\Upsilon/X]}\ \text{(linear type substitution)}\ .$$

Suppose $\Theta \vdash !^n(\forall X{<:}\Phi_X.\Psi)$ has been derived by

$$\frac{\Xi, \Xi' \vdash \Phi_X <: !\mathit{Top} \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash !^n\Psi}{\Xi, \Xi' \vdash !^n(\forall X{<:}\Phi_X.\Psi)}\ \text{(nonlinear-polymorphic type)}\ .$$

As before, we know $\Xi, X{<:}\Phi_X$ is consistent. Using Lemma 4.19, we find $\Theta = \Xi, \Xi'$ to be consistent, as well. Type weakening and type permutation then yield proved type $\Theta, X{<:}\Phi_X \vdash !^n\Psi$ analogously to the previous case. From proved subtypes $\Xi, \Xi' \vdash \Phi_X <: !\mathit{Top}$ and $\Theta \vdash \Upsilon <: \Phi_X$ and Corollary 4.17 deduce we know $\Phi_X = !\Phi'_X$ and $\Upsilon = !\Upsilon'$ hold for certain type terms Φ'_X and Υ'. Using rule (! amplification), we get proved type $\Theta \vdash !!\Upsilon' = \Theta \vdash !\Upsilon$ from $\Theta \vdash !\Upsilon'$. Now we can use rule (nonlinear type substitution) to derive $\Theta \vdash !^n\Psi[!\Upsilon'/X]$ by

$$\frac{\Theta, X{<:}\Phi_X \vdash !^n\Psi \qquad \begin{array}{c}\Theta \vdash !\Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array}}{\Theta \vdash !^n\Psi[\Upsilon/X]}\ \text{(nonlinear type substitution)}\ .$$

Hence, we have shown proved type $\Theta \vdash !^n\Psi[\Upsilon/X]$ can be derived in both cases, which concludes the proof. □

So far, we have considered type terms and proved types in isolation. However, a type system does usually not exist in isolation, but rather is constructed to be used as a tool to allow for drawing conclusions about programs or, to be more precise, function terms in case of *QLC*. At this point we do not only finish our investigation of type preservation at an isolated type level, but in general give up our narrowed focus in order to widen it in the next section to also cover function terms. In what follows we extend the function terms introduced in section 3 so that they involve additional syntactic constructs for type abstraction and type application, and we adapt operational semantics accordingly to also realize the concept of type application as a form of term reduction. In addition, and even more interesting and important, we interweave our type system with *QLC* function terms. To achieve this, we also give derivation rules for proved terms, which facilitate the assignment of types to function terms. Afterwards, we investigate the implications that follow from these rules and finally return to the type preservation property as one corner stone of type safety for polymorphically typed *QLC*.

4.3 Function terms

In the course of this section we lift untyped *QLC* from section 3 to a polymorphically typed calculus. We have already developed and investigated the foundations of polymorphically typed *QLC* on the level of types in the previous subsections. It thus remains to join the concepts and notions comprising untyped *QLC* and the polymorphic type system from section 4.2 to form polymorphically typed *QLC*. Although we do not present simply typed *QLC* in the present work, we still base our polymorphic extension as far as possible on simply typed *QLC* as it is defined in [SV09]. This intended proximity also motivates some design choices we make, for instance the restriction of the used subtype relation in subsection 4.3.2. On the other hand, we take the freedom of modifying the syntactic presentation a bit, as we have already done when presenting untyped *QLC*. Moreover, we intend to keep most of simply typed *QLC*'s characteristics in tact, especially concerning type safety. To which extend we reach this goal will be further investigated in section 4.6.

4.3.1 Basic definitions

As usual, we start with the very basic notions and properties. Most of the following definitions directly extend definitions from section 3.

Definition 4.51 (polymorphically typed *QLC* function terms, bound term variables).
As in Definition 3.1, sets $\mathcal{V}_{term}$ and $\mathcal{C}_{term}$ stand for term variables and *QLC* term constants, respectively. We inductively define the set $\mathcal{T}_{pQLC}$ of *polymorphically typed QLC function terms* (or just *function terms*, when there is no danger of confusion with untyped *QLC* function terms):
term variables x, term constants c, term application $(s\ t)$, pair terms $\langle t_1, t_2\rangle$, the empty tuple $\langle\rangle$ and disjoint unions $inj_l(t)$ and $inj_r(t)$ are defined analogously to Definition 3.1. The remaining syntactic constructs are modified in the following way:

- term abstraction:
  $(\lambda x{:}\Phi_x.t) \in \mathcal{T}_{pQLC}$
  for all $x \in \mathcal{V}_{term}$; $\Phi_x \in \mathcal{T}_{type}$ and $t \in \mathcal{T}_{pQLC}$,
- pair abstraction:
  $\big(\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t\big) \in \mathcal{T}_{pQLC}$
  for all $x, y \in \mathcal{V}_{term}$ with $x \neq y$; $\Phi_x, \Phi_y \in \mathcal{T}_{type}$ and $t \in \mathcal{T}_{pQLC}$,
- case distinction:
  $\big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big) \in \mathcal{T}_{pQLC}$
  for all $x, y \in \mathcal{V}_{term}$; $\Phi_x, \Phi_y \in \mathcal{T}_{type}$ and $s, t_l, t_r \in \mathcal{T}_{pQLC}$,
- recursion term:
  $\big(letrec\ f{:}\Phi_f = (\lambda x{:}\Phi_x.s)\ in\ t\big) \in \mathcal{T}_{pQLC}$
  for all $f, x \in \mathcal{V}_{term}$; $\Phi_f, \Phi_x \in \mathcal{T}_{type}$ and $s, t \in \mathcal{T}_{pQLC}$.

Moreover, we introduce two new syntactic constructs:

- type abstraction:
  $\big(\Lambda X{<:}\Phi_X.t\big) \in \mathcal{T}_{pQLC}$
  for all $X \in \mathcal{V}_{type}$; $\Phi_X \in \mathcal{T}_{type}$ and $t \in \mathcal{T}_{pQLC}$,

- type application:
  $\big(s\ \Phi\big) \in \mathcal{T}_{pQLC}$
  for all $s \in \mathcal{T}_{pQLC}$ and $\Phi \in \mathcal{T}_{type}$,

The notions of *scopes*, *bound* and *free* occurrences of term variables are adapted accordingly. Furthermore, we call t the *scope* of a type abstraction $(\Lambda X{<:}\Phi_X.t)$, and every free occurrence of X in t is considered to be *bound* by this type abstraction.

Most of the syntactic constructs defined in this definition are familiar to us from untyped *QLC*. Some have been modified to carry annotations indicating the type of bound term or type variables. One fact immediately catching our eye is this use of explicit typing. This is in contrast to simply typed *QLC* in [SV09] where function terms are implicitly typed, which means they are written without annotations regarding the type of term variable x in a term abstraction $(\lambda x{:}\Phi_x.t)$, for instance. In the present work we choose explicit typing, although we actually follow the Curry style of language definition, i.e. "Semantics is prior to typing" ([Pie02], page 111).[43]

Remark: That we pursue the Curry style of language definition is evidently so because when we lift operational semantics from the untyped setting to polymorphically typed *QLC* we obtain a semantics in terms of reduction of function terms that also works on function terms t for which we cannot derive a proved term $\Theta|\Gamma \vdash t : \Phi$ (cf. Definition 4.61). From the perspective of term reduction such function terms do not pose an obstacle. If t is irreducible and not a value term, we consider it as an *error term* (cf. Definition 4.56). On the other hand, a reducible t might model a computation in the quantum world that does actually violate physical law. Our approach is to rule out such function terms by means of typing. But the operational semantics we define in section 4.5 is in fact oblivious to physical law, and thus also allows reduction of function terms to which we cannot necessarily assign a proper type.

In contrast to untyped *QLC*, the only newly introduced syntactic constructs for formation of polymorphically typed function terms are type abstraction and type application. The underlying concepts go back to Girard's system *F*.[44] However, we here combine the standard concepts of second-order typing with the concept of subtyping and end up with bounded quantification, as it is presented for instance in chapter 26 in [Pie02] and also in [CMM91]. In such type abstractions we annotate the type bound Φ_X of bound type variable X as well, but here we informally mean we can apply any type term Υ to this type abstraction in a type application $((\Lambda X{<:}\Phi_X.t)\ \Upsilon)$ as long as Υ is a subtype of Φ_X. This restriction, however, is not checked or enforced on the syntactic level, but we deal with it at the level of typed terms during derivations of proved terms in section 4.4.

We have already encountered type abstractions in the previous section and have also already been referring to the concept of type application, but on an informal level. The underlying computational mechanisms (which we introduce in section 4.5) are analogous to the ones concerned with term application, i.e. they are based on substitution of free variables, and are thus quite familiar to us. More interesting, however, are the issues that are connected with the derivation of proved terms with type application. This will come into the focus of our attention soon.

But before we arrive at the more interesting core part of our polymorphic type system, we first extend the notions of free occurrences of term variables and substitution of free term variables and free type variables to the polymorphic setting.

Definition 4.52 (free term variables, free type variables, α-equivalence).
The notion of *free term variables* for typed function terms follows the same intention as in Definition 3.2, and hence the function *ftmv* is adapted according to the modified syntax of polymorphic function terms to obtain $ftmv : \mathcal{T}_{pQLC} \to \mathcal{P}(\mathcal{V}_{term})$. However, we here explicitly give the definition

[43] For a clarifying discussion on the terminology of implicit and explicit typing and their relationship to "Curry-style" and "Church-style" language definitions, see section 9.6 in [Pie02].

[44] For theoretical investigations on system *F* see chapter 11 in [GTL90] for a treatment in the context of proof theory and typing, or [Bar92] for an overview of system *F* (therein mostly called $\lambda 2$) in its Curry-style and Church-style variant in the context of typed λ-calculi and Barendregt's famous λ-cube. For a more practically minded introduction to system *F* in the context of typing for programming languages, see chapter 23 in [Pie02].

of function *ftmv* on the new syntactic constructs:

$$\begin{aligned} \mathit{ftmv}\big((\Lambda X{<:}\Phi_X.t)\big) &:= \mathit{ftmv}(t)\ , \\ \mathit{ftmv}\big((s\ \Phi)\big) &:= \mathit{ftmv}(s)\ . \end{aligned}$$

We furthermore extend function $\mathit{ftyv} : \mathcal{T}_{type} \to \mathcal{P}(\mathcal{V}_{type})$ from Definition 4.3 to function $\mathit{ftyv} : \mathcal{T}_{pQLC} \cup \mathcal{T}_{type} \to \mathcal{P}(\mathcal{V}_{type})$, which yields the set $\mathit{ftyv}(t)$ of all type variables that occur as *free type variables* in function term t. We directly take over the definition of *ftyv* on type terms and extend it as follows to also handle function terms (for $c \in \mathcal{C}_{term}$ and $x \in \mathcal{V}_{term}$):

$$\begin{aligned} \mathit{ftyv}\big(c\big) &:= \emptyset\ , \\ \mathit{ftyv}\big(x\big) &:= \emptyset\ , \\ \mathit{ftyv}\big((\lambda x{:}\Phi_x.t)\big) &:= \mathit{ftyv}(\Phi_x) \cup \mathit{ftyv}(t)\ , \\ \mathit{ftyv}\big((s\ t)\big) &:= \mathit{ftyv}(s) \cup \mathit{ftyv}(t)\ , \\ \mathit{ftyv}\big(\langle t_1, t_2\rangle\big) &:= \mathit{ftyv}(t_1) \cup \mathit{ftyv}(t_2)\ , \\ \mathit{ftyv}\big(\langle\rangle\big) &:= \emptyset\ , \\ \mathit{ftyv}\big((\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t)\big) &:= \mathit{ftyv}(\Phi_x) \cup \mathit{ftyv}(\Phi_y) \cup \mathit{ftyv}(t)\ , \\ \mathit{ftyv}\big(\mathit{inj}_l(t)\big) = \mathit{ftyv}\big(\mathit{inj}_r(t)\big) &:= \mathit{ftyv}(t)\ , \\ \mathit{ftyv}\big((\mathit{match}\ s\ \mathit{with}\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r))\big) &:= \mathit{ftyv}(s) \cup \mathit{ftyv}((\lambda x{:}\Phi_x.t_l)) \cup \mathit{ftyv}((\lambda y{:}\Phi_y.t_r))\ , \\ \mathit{ftyv}\big((\mathit{letrec}\ f{:}\Phi_f = (\lambda x{:}\Phi_x.s)\ \mathit{in}\ t)\big) &:= \mathit{ftyv}(\Phi_f) \cup \mathit{ftyv}((\lambda x{:}\Phi_x.s)) \cup \mathit{ftyv}(t)\ , \\ \mathit{ftyv}\big((\Lambda X{<:}\Phi_X.t)\big) &:= \mathit{ftyv}(\Phi_X) \cup \big(\mathit{ftyv}(t) \setminus \{X\}\big)\ , \\ \mathit{ftyv}\big((s\ \Phi)\big) &:= \mathit{ftyv}(s) \cup \mathit{ftyv}(\Phi)\ . \end{aligned}$$

We call function terms *α-equivalent* if they differ only in the names of their bound type and term variables, i.e. if they have the same structure.

As in the previous chapters, we follow the usual convention to identify α-equivalent function terms.

Definition 4.53 (substitution of type variables and term variables).
We adapt the definition of *substitution of free term variables* from Definition 3.3 to the setting of polymorphically typed function terms in accordance to the above definition of set $\mathcal{T}_{pQLC}$. For the newly introduced syntactic constructs, we define

$$\begin{aligned} (\Lambda X{<:}\Phi_X.t)[t'/z] &:= (\Lambda X{<:}\Phi_X.t[t'/z])\ , \\ (s\ \Phi)[t'/z] &:= (s[t'/z]\ \Phi)\ . \end{aligned}$$

Moreover, we extend the formal mechanism of *substitution of free type variables* to be applicable to typed function terms as well:

$$\begin{aligned} c[\Upsilon/Z] &:= c\ , \\ x[\Upsilon/Z] &:= x\ , \\ \big(\lambda x{:}\Phi_x.t\big)[\Upsilon/Z] &:= \big(\lambda x{:}\Phi_x[\Upsilon/Z].t[\Upsilon/Z]\big)\ , \\ \big(s\ t\big)[\Upsilon/Z] &:= \big(s[\Upsilon/Z]\ t[\Upsilon/Z]\big)\ , \\ \big\langle t_1, t_2\big\rangle[\Upsilon/Z] &:= \big\langle t_1[\Upsilon/Z], t_2[\Upsilon/Z]\big\rangle\ , \\ \big\langle\big\rangle[\Upsilon/Z] &:= \big\langle\big\rangle\ , \\ \big(\lambda\big\langle x{:}\Phi_x, y{:}\Phi_y\big\rangle.t\big)[\Upsilon/Z] &:= \big(\lambda\big\langle x{:}\Phi_x[\Upsilon/Z], y{:}\Phi_y[\Upsilon/Z]\big\rangle.t[\Upsilon/Z]\big)\ , \\ \mathit{inj}_l\big(t\big)[\Upsilon/Z] &:= \mathit{inj}_l\big(t[\Upsilon/Z]\big)\ , \\ \mathit{inj}_r\big(t\big)[\Upsilon/Z] &:= \mathit{inj}_r\big(t[\Upsilon/Z]\big)\ , \\ \big(\mathit{match}\ s\ \mathit{with}\ \big(\lambda x{:}\Phi_x.t_l\big) \mid \big(\lambda y{:}\Phi_y.t_r\big)\big)[\Upsilon/Z] &:= \\ \big(\mathit{match}\ s[\Upsilon/Z]\ \mathit{with}\ \big(\lambda x{:}\Phi_x[\Upsilon/Z].t_l[\Upsilon/Z]\big) \mid \big(\lambda y{:}\Phi_y[\Upsilon/Z].t_r[\Upsilon/Z]\big)\big)\ , \\ \big(\mathit{letrec}\ f{:}\Phi_f = \big(\lambda x{:}\Phi_x.s\big)\ \mathit{in}\ t\big)[\Upsilon/Z] &:= \\ \big(\mathit{letrec}\ f{:}\Phi_f[\Upsilon/Z] = \big(\lambda x{:}\Phi_x[\Upsilon/Z].s[\Upsilon/Z]\big)\ \mathit{in}\ t[\Upsilon/Z]\big)\ , \end{aligned}$$

$$\begin{aligned}
\big(\Lambda Z{<:}\Phi_Z.t\big)[\Upsilon/Z] &:= (\Lambda Z{<:}\Phi_Z[\Upsilon/Z].t)\ ,\\
\big(\Lambda X{<:}\Phi_X.t\big)[\Upsilon/Z] &:= \big(\Lambda X'{<:}\Phi_X[\Upsilon/Z].t[X'/X][\Upsilon/Z]\big)\ ,\\
&\qquad\qquad \text{where } Z \neq X \text{ and } X' \notin \mathit{ftyv}(t) \cup \mathit{ftyv}(\Upsilon) \cup \{X, Z\},\\
\big(s\ \Phi\big)[\Upsilon/Z] &:= \big(s[\Upsilon/Z]\ \Phi[\Upsilon/Z]\big)\ .
\end{aligned}$$

Interestingly, we do not need to extend function *nftyv* to work on set $\mathcal{T}_{pQLC}$ because it is sufficient for our investigations to consider nonlinear free type variables in type terms only. Hence, at this point we are finished with extending the basic concepts we have already seen in the section on untyped *QLC* to the setting of polymorphically typed *QLC*. Now we can start working out the parts of our extension that are essentially new compared to the untyped variant.

4.3.2 The restricted subtype relation $\prec$:

When we introduced type terms and proved types, we have defined the subtype relation $<:$ as the smallest relation fulfilling axioms (1) to (8) from Definition 4.7. We have confirmed later (cf. subsection 4.1.3) that our subtype relation is an extension of the one used in [SV09]. One particularity of our subtype relation is type constant *Top* which we use to obtain a type that is supertype of all other types. *Top* is a convenient tool to enforce duplicability of type terms that may be applied to certain type abstractions. We will see later how this works, namely when we formulate derivation rules for proved terms.

On the other hand, we said we want to stay as close to simply typed *QLC* as possible. However, against this aim stands type constant *Top* as a supertype of all other types, since it brings along a degree of freedom in the derivation of proved subtypes that indeed goes too far beyond the capabilities of simply typed *QLC*'s type system. Thus, we need to decide whether we really want to substantially extend the possibilities of type assignment that we have for function terms in the polymorphically typed setting in comparison to simply typed *QLC*. The decision we make is to restrict specific derivations of proved subtypes reasonably, in order to not allow *Top* to appear in proved subtypes in certain places. However, we will discuss the circumstances that lead to this choice later (after Definition 4.63 and also after Definition 4.62), since we are not yet in the right position to formulate our arguments in a clear way.

For now we just formulate the restriction of subtype derivations and take a look at an interesting consequence of this definition.

Definition 4.54 (restricted subtype relation and restricted proved subtypes).
We define the *restricted subtype relation* $\prec:$ to be the smallest binary relation on $\mathcal{T}_{type}$ satisfying axioms (1) and (3) to (8) given in Definition 4.7.

We call a proved subtype $\Theta \vdash \Phi <: \Psi$ a *restricted proved subtype* if it can be derived without the use of derivation rule (*Top* supertype). To indicate that a proved subtype $\Theta \vdash \Phi <: \Psi$ is a restricted proved subtype, we write $\Theta \vdash \Phi \prec: \Psi$.

The more important part of this definition clearly is the second one. It is so important, since we get the following property of restricted proved subtypes from it:

Proposition 4.55. *Let $\Theta \vdash \Phi \prec: \Psi$ be a restricted proved subtype. Then Φ and Ψ differ only in the occurrences of exponentials, i.e. they have the same term structure as soon as we remove all exponentials from both Φ and Ψ.*

Proof. Consider a restricted proved subtype $\Theta \vdash \Phi \prec: \Psi$ and recall the subterm addressing function *sta* from Definition 4.44. We show by induction on the derivation of $\Theta \vdash \Phi \prec: \Psi$ that the addresses associated to subterms in $\mathit{sta}(\Phi)$ and $\mathit{sta}(\Psi)$ have the same structure, i.e. for each $(p, \Sigma_\Phi) \in \mathit{sta}(\Phi)$, there exists a unique $(p, \Sigma_\Psi) \in \mathit{sta}(\Psi)$ and vice versa, where type terms Σ_Φ and Σ_Ψ differ only in the occurrences of exponentials.

Base case: If $\Theta \vdash \Phi \prec: \Psi$ has been derived by ($<:$ reflexivity), then we know $\Phi = \Psi$ and thus also $\mathit{sta}(\Phi) = \mathit{sta}(\Psi)$.

Induction cases: Suppose $\Theta \vdash \Phi \prec: \Psi$ has been derived by

$$\frac{\Theta \vdash \Phi' \prec: \Psi}{\Theta \vdash !\Phi' \prec: \Psi} \text{ (! left)} .$$

By definition of function *sta* we then have

- $(\varepsilon, \Sigma_\Phi) \in sta(!\Phi')$ implies $\Sigma_\Phi = !^{n+1}\Phi''$,
- $(\varepsilon, \Sigma'_\Phi) \in sta(\Phi')$ implies $\Sigma'_\Phi = !^n\Phi''$ and
- $sta(!\Phi') \setminus \{(\varepsilon, !^{n+1}\Phi'')\} = sta(\Phi') \setminus \{(\varepsilon, !^n\Phi'')\}$,

for some type term Φ'' and $n \geq 0$. Then induction yields there is a unique $(p, \Sigma_\Psi) \in sta(\Psi)$ for each $(p, \Sigma'_\Phi) \in sta(\Phi')$ and thus also for each $(p, \Sigma_\Phi) \in sta(!\Phi')$ and vice versa, and for each such unique triple $(p, \Sigma_\Psi), (p, \Sigma'_\Phi), (p, \Sigma_\Phi)$ type terms Σ_Φ and Σ_Ψ differ only in the occurrences of exponentials. The case of derivation rule (! right) is symmetrical.

Suppose $\Theta \vdash \Phi \prec: \Psi$ has been derived by

$$\frac{\Theta \vdash \Psi_1 \prec: \Phi_1 \qquad \Theta \vdash \Phi_2 \prec: \Psi_2}{\Theta \vdash (\Phi_1 \multimap \Phi_2) \prec: (\Psi_1 \multimap \Psi_2)} \text{ (function subtype)} .$$

By induction we then get

- for each $(p, \Sigma_{\Psi_1}) \in sta(\Psi_1)$ there exists a unique $(p, \Sigma_{\Phi_1}) \in sta(\Phi_1)$, and vice versa, and for each such unique pair $(p, \Sigma_{\Psi_1}), (p, \Sigma_{\Phi_1})$ type terms Σ_{Ψ_1} and Σ_{Φ_1} differ only in the occurrences of exponentials;
- for each $(p, \Sigma_{\Phi_2}) \in sta(\Phi_2)$ there exists a unique $(p, \Sigma_{\Psi_2}) \in sta(\Psi_2)$, and vice versa, and for each such unique pair $(p, \Sigma_{\Phi_2}), (p, \Sigma_{\Psi_2})$ type terms Σ_{Φ_2} and Σ_{Ψ_2} differ only in the occurrences of exponentials.

Let us define the following four sets

$$\begin{aligned} s_{\Phi_1} &:= \big\{(1p, \Sigma_{\Phi_1}) \;\big|\; (p, \Sigma_{\Phi_1}) \in sta(\Phi_1)\big\} , \\ s_{\Phi_2} &:= \big\{(2p, \Sigma_{\Phi_2}) \;\big|\; (p, \Sigma_{\Phi_2}) \in sta(\Phi_2)\big\} , \\ s_{\Psi_1} &:= \big\{(1p, \Sigma_{\Psi_1}) \;\big|\; (p, \Sigma_{\Psi_1}) \in sta(\Psi_1)\big\} , \\ s_{\Psi_2} &:= \big\{(2p, \Sigma_{\Psi_2}) \;\big|\; (p, \Sigma_{\Psi_2}) \in sta(\Psi_2)\big\} . \end{aligned}$$

Consequently, for each

$$(p, \Sigma_\Phi) \in sta\big((\Phi_1 \multimap \Phi_2)\big) = \big\{\big(\varepsilon, (\Phi_1 \multimap \Phi_2)\big)\big\} \cup s_{\Phi_1} \cup s_{\Phi_2}$$

there exists a unique

$$(p, \Sigma_\Psi) \in sta\big((\Psi_1 \multimap \Psi_2)\big) = \big\{\big(\varepsilon, (\Psi_1 \multimap \Psi_2)\big)\big\} \cup s_{\Psi_1} \cup s_{\Psi_2} ,$$

and vice versa, and for each such unique pair $(p, \Sigma_\Phi), (p, \Sigma_\Psi)$ type terms Σ_Φ and Σ_Ψ differ only in the occurrences of exponentials.
The remaining induction cases can be argued along similar lines.

□

By this proposition we now have confirmed that if $\Theta \vdash \Phi \prec: \Psi$ is a restricted proved subtype, then Φ and Ψ have the same structure except for occurring exponentials in both type terms. We discuss the benefits of restricted subtype derivations in the course of the next section. Furthermore, we will see later, especially in the parts concerned with type preservation, that some of our results indeed depend on such a restriction. If we did not restrict subtype derivations in certain places, we would have to take other measures with farther-reaching consequences to still ensure type preservation for polymorphically typed *QLC*. But the mentioned consequences include possibilities of type assignments that reach too far beyond the capabilities of simply typed *QLC* so that we do not accept them for our present work. On the other hand, they may very well be of interest for future investigations.

4.4 Proved terms

On the level of types we use derivation rules to distinguish a special subset of type terms that we consider well-formed, namely the ones, say Φ, for which we find a proper type context Θ, so that we can derive proved type $\Theta \vdash \Phi$. In this subsection we use the same approach to rule out function terms that do not make sense from two points of view. The first one is that of linearity as we have discussed it in the beginning of section 4, where we argued the impact of the no-cloning theorem (Theorem 2.5) on formation of *QLC* function terms. The second point of view is that of error terms.

Definition 4.56 (error states, error terms).
An *error state* is represented by an irreducible quantum closure $[Q, L, e]$ where e is not a value term. The embedded function term e (which is thus also irreducible) is then called *error term*.

Let us take a short look at some examples of error terms:

- $\big(U_2\ \langle q_1, q_1\rangle\big)$ with $U_2 \in \mathcal{U}_2$ is irreducible due to the two indistinct components in the pair,[45]
- $\big(U_2\ \langle q_1, q_2, q_3\rangle\big)$ with $U_2 \in \mathcal{U}_2$ is irreducible, since we try to apply a binary built-in unitary operator to a triple of qubits,
- $\big((\lambda\langle x, y\rangle.x\ y)\ (\lambda z.z)\big)$ is irreducible, since $(\lambda z.z)$ is not a pair term,
- $\big(match\ (\lambda z.z)\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)\big)$ is irreducible, since term abstraction $(\lambda z.z)$ is in a place where operational semantics expects a disjoint union,
- ... .

What Selinger and Valiron achieve in simply typed *QLC* is that they rule out error terms with their type system. And of course, we intend to keep this very nice feature also in our polymorphic extension. We investigate in section 4.6 whether this property named *type safety* actually is preserved in our type system.

4.4.1 Basic definitions and properties

In analogy to proved types we first need to define some basic notions before we come to the derivation rules for proved terms. Again, the notion of consistency plays an important role, however, in a slightly different way.

At first, we need a concept of a typing environment for free term variables.

Definition 4.57 (term contexts and related notions).
A *term context* is a finite, possibly empty set of variable-type pairs, denoted $\Gamma = x_1{:}\Phi_1, \ldots, x_n{:}\Phi_n$, where all $x_1, \ldots, x_n$ are required to be distinct.

We write $|\Gamma|$ to denote the set of all term variables that appear left of a colon in term context Γ, and sometimes refer to $|\Gamma|$ as the *domain* of Γ. Moreover, we write $\Gamma(x_i)$ to denote the type term Φ_i assigned to x_i in Γ, for all i with $1 \leq i \leq n$.

The *union of two term contexts* Γ and Γ' is denoted as juxtaposition Γ, Γ' if and only if their domains $|\Gamma|$ and $|\Gamma'|$ are disjoint, i.e. $|\Gamma| \cap |\Gamma'| = \emptyset$.

The restriction of type terms right of a colon in a term context Δ to duplicable type terms only is denoted as $!\Delta$. That means, $!\Delta = x_1{:}\Phi_1, \ldots, x_n{:}\Phi_n$ implies for each i with $1 \leq i \leq n$ that $\Phi_i = !\Phi_i'$ holds for some $\Phi_i' \in \mathcal{T}_{type}$.

An important difference of this definition compared to the definition of type contexts is that term contexts are sets and not sequences. That means there is no specific order of variable-type pairs in a term context Γ, although we occasionally write it down as sequence $\Gamma = x_1{:}\Phi_1, \ldots, x_n{:}\Phi_n$ for convenience. The treatment of term contexts as sets is possible since a term variable $x \in |\Gamma|$ that is being assigned type term $\Gamma(x)$ in Γ does not play any role in type terms $\Gamma(y)$ for all $y \in |\Gamma| \setminus \{x\}$.

[45] Such function terms cannot be evaluated because the application of a unitary operator of arity 2 to a single qubit in the sense function term $(U_2\ \langle q_1, q_1\rangle)$ implies cannot be formulated in the formal framework of quantum computation (cf. section 2).

But there is another consequence of this observation: notions of well-scopedness and consistency as we have them for type contexts do not make much sense for term contexts. However, we still have a concept of consistency, but as already announced earlier, this one comes in a slightly different flavor.

Definition 4.58 (consistency of a term context).
Let Θ be a consistent type context. A term context $\Gamma = x_1{:}\Phi_1, \ldots, x_n{:}\Phi_n$ is called *consistent with respect to* Θ, denoted as $\Theta \vdash \Gamma$, if $\Theta \vdash \Phi_1, \ldots, \Theta \vdash \Phi_n$ are proved types. In case of an empty type context Θ, we simply write $\vdash \Gamma$.

In particular, the empty term context is considered to be consistent with respect to any consistent type context Θ. In this case $\Theta \vdash \emptyset$ translates to $\vdash \Theta$.

It is quite clear what we aim at with this notion of consistency. With its help we can achieve that only type terms Φ for which there exists a proved type $\Theta \vdash \Phi$ may be assigned to free term variables in term contexts. The bitter pill we have to swallow for this is, however, a more complicated and technical looking syntax for typing judgements denoting proved terms, and the same for the respective derivation rules.

Definition 4.59 (term-in-context).
A judgement $\Theta|\Gamma \vdash t : \Phi$ (where Θ is a type context, Γ is a term context, $t \in \mathcal{T}_{pQLC}$ is a function term and $\Phi \in \mathcal{T}_{type}$ is a type term) is called *term-in-context*.

For convenience, we simply write $\Gamma \vdash t : \Phi$ if the type context is empty. If the term context is empty, we write $\Theta|\emptyset \vdash t : \Phi$. And finally, we write $\vdash t : \Phi$ if both contexts are empty.

At this point we have almost readily set up the stage for the centerpiece of section 4.3. But before we come to it, we agree on three additional conventions that we will be using henceforth.

Convention 4.60.

(i) We use the following abbreviation: $\mathit{Bit} := (\mathit{Unit} \oplus \mathit{Unit})$.

(ii) Let $\Gamma = x_1{:}\Phi_1, \ldots, x_n{:}\Phi_n$ be a term context. We write $\mathit{ftyv}(\Gamma)$ to mean $\bigcup_{i=1}^{n} \mathit{ftyv}(\Phi_i)$.

(iii) In analogy to an abbreviation we have already been introducing in section 2, we use the notation $\Phi^{\otimes n}$ to mean $(\Phi \otimes (\Phi \otimes \ldots (\Phi \otimes \Phi)\ldots))$ where type term Φ shall occur n times.

In Definitions 4.62 to 4.65 we give all derivation rules for proved terms, separated in four groups. In between we discuss the form of the rules and some noteworthy peculiarities.

Definition 4.61 (proved terms in general, α-equivalence).
A *proved term* is a term-in-context that can be derived using the derivation rules given in Definitions 4.62 to 4.65, where we assume $m, n \geq 0$.

We moreover consider proved terms $\Theta|\Gamma \vdash t : \Phi$ and $\Theta'|\Gamma' \vdash t' : \Phi'$ to be *α-equivalent* if the following conditions are fulfilled:

- type contexts Θ and Θ' are α-equivalent,
- $|\Gamma| = |\Gamma'|$ and $\Gamma(x) \equiv_\alpha \Gamma'(x)$ for all $x \in |\Gamma|$,
- function terms t and t' are α-equivalent, and
- type terms Φ and Φ' are α-equivalent.

As usual, we henceforth identify α-equivalent proved terms.

Now we come to the definition of derivation rules for proved terms, starting with the rules for term constants from $\mathcal{C}_{term}$.

Definition 4.62 (proved terms – term constants).

$$\frac{\Theta \vdash \Gamma \qquad \begin{array}{c}\Theta \vdash \Psi \\ \Theta \vdash !(\mathit{Bit} \multimap \mathit{Qbit}) \prec: \Psi\end{array}}{\Theta|\Gamma \vdash \mathrm{new} : \Psi}\ (\text{new operator})$$

$$\frac{\Theta \vdash \Psi \qquad \Theta \vdash \Gamma \qquad \Theta \vdash !(\mathit{Qbit} \multimap !\mathit{Bit}) \prec: \Psi}{\Theta|\Gamma \vdash \mathrm{meas} : \Psi} \text{ (meas operator)}$$

$$\frac{\Theta \vdash \Psi \qquad \Theta \vdash \Gamma \qquad \Theta \vdash !(\mathit{Qbit}^{\otimes m+1} \multimap \mathit{Qbit}^{\otimes m+1}) \prec: \Psi \qquad U \in \mathcal{U}_{m+1}}{\Theta|\Gamma \vdash U : \Psi} \text{ (unitary operator)}$$

These derivation rules could be counted as axioms (together with rule (term variable) that will follow in Definition 4.63 and together with (*Unit* term) in Definition 4.64) since they do themselves not depend on a proved term. The above rules are special in a number of different ways.

Firstly, they contain a fixed minimal (with respect to $<:$) type for operators *new* and *meas* and also for built-in unitary operators. So, for instance, we have fixed that the type of *new* can be any supertype of $!(\mathit{Bit} \multimap \mathit{Qbit})$ which we can derive without using rule (*Top* supertype). We moreover fix that a built-in unitary operator $U \in \mathcal{U}_{m+1}$ maps $(m+1)$-tuples of function terms of type *Qbit* to such $(m+1)$-tuples again. This fully corresponds to the respective probabilistic reduction rule in Definition 3.10 (which we lift to the polymorphic case in section 4.5). Please further note we may assign a duplicable type to all these operators. Hence, in accordance to our intuition, they may appear in different places in function terms at the same time.

Secondly, the above derivation rules ensure consistency of term context Γ with respect to type context Θ in their first premise. This lays the foundation of our later consistency results in Lemma 4.66.

Thirdly, the check for $\Theta \vdash \Psi$ being a proved type is necessary to prevent types such as $(!\mathit{Qbit} \multimap \mathit{Bit})$ in rule (meas operator), for instance. We have already discussed earlier that a proved subtype $\Theta \vdash \Phi <: \Psi$ does not facilitate any conclusions about whether $\Theta \vdash \Phi$ or $\Theta \vdash \Psi$ are proved types or whether they are not. If we did not include this check among the premises of the above rules, we might corrupt results such as "If $\Theta|\Gamma \vdash t : \Phi$ is a proved function term, then $\Theta \vdash \Phi$ is a proved type" (cf. Lemma 4.66). Similar reasons motivate similar checks in rules (term variable) and (type application).

Fourthly, we restrict subtype derivations to not use rule (*Top* supertype) in the third premises. Simply typed *QLC* does not allow the assignment of a type *Top* to a function term, or even a type that contains *Top* as a subterm. On the one hand, it would thus be a quite substantial change of the capabilities of *QLC* if we introduced the *Top* type. On the other hand, this might be a quite desirable opportunity to introduce new language features into *QLC*. However, this lies clearly beyond the scope of the present work. The problems is, such a change of capabilities would require the additional introduction of at least one new derivation rule to resolve some technical issues connected with type preservation. The mentioned additional rule has (in principle) the form[46]

$$\frac{\Theta \vdash \Psi \qquad \Theta|\Gamma \vdash t : \Phi \qquad \Theta \vdash \Phi <: \Psi}{\Theta|\Gamma \vdash t : \Psi} \ .$$

Introducing such a derivation rule might entail further consequences that we would need to explore, but, as already said, this is beyond the scope of the present work and might be subject to future investigations.

The just discussed restriction of subtype derivations has the same motivation also in rule (term variable) in the following definition.

Definition 4.63 (proved terms – core rules).

$$\frac{\Theta \vdash \Psi_x \qquad \Theta \vdash \Gamma, x{:}\Phi_x \qquad \Theta \vdash \Phi_x \prec: \Psi_x}{\Theta|\Gamma, x{:}\Phi_x \vdash x : \Psi_x} \text{ (term variable)}$$

$$\frac{\Theta|\Gamma, x{:}\Phi_x \vdash t : \Psi}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t) : (\Phi_x \multimap \Psi)} \text{ (linear term abstraction)}$$

$$\frac{\Theta|!\Delta, \Gamma, x{:}\Phi_x \vdash t : \Psi \qquad \mathit{ftmv}(t) \cap |\Gamma| = \emptyset}{\Theta|!\Delta, \Gamma \vdash (\lambda x{:}\Phi_x.t) : !^{n+1}(\Phi_x \multimap \Psi)} \text{ (nonlinear term abstraction)}$$

[46] This rule has been inspired by rule (T-SUB) from Figure 26-1 on page 392 in [Pie02].

$$\frac{\Theta|!\Delta,\Gamma_1 \vdash s : !^n(\Phi \multimap \Psi) \qquad \Theta|!\Delta,\Gamma_2 \vdash t : \Phi}{\Theta|!\Delta,\Gamma_1,\Gamma_2 \vdash (s\ t) : \Psi}\ \text{(term application)}$$

$$\frac{\vdash \Theta,\Theta' \qquad \Theta, X{<:}\Phi_X, \Theta'|\Gamma \vdash t : \Psi \qquad \begin{array}{c} X \notin \mathit{nftyv}(\Psi) \\ X \notin \mathit{ftyv}(\Gamma) \end{array}}{\Theta,\Theta'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : (\forall X{<:}\Phi_X.\Psi)}\ \text{(linear type abstraction)}$$

$$\frac{\Theta,\Theta' \vdash \Phi_X <: !\mathit{Top} \qquad \Theta, X{<:}\Phi_X, \Theta'|\Gamma \vdash t : !^n\Psi \qquad X \notin \mathit{ftyv}(\Gamma)}{\Theta,\Theta'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : !^n(\forall X{<:}\Phi_X.\Psi)}\ \text{(nonlinear type abstraction)}$$

$$\frac{\Theta|\Gamma \vdash t : !^n(\forall X{<:}\Phi_X.\Psi) \qquad \begin{array}{c} \Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X \end{array}}{\Theta|\Gamma \vdash (t\ \Upsilon) : !^n\Psi[\Upsilon/X]}\ \text{(type application)}$$

The just defined set of rules is essentially the heart of our polymorphic type system. The treated syntactic constructs are already rich enough to form a calculus on their own that would be resembling system F with incorporated linearity, subtyping and bounded quantification.

Let us take a closer look at rule (term variable). It is quite similar to the derivation rules concerned with type constants. However, the third premise has farther-reaching consequences. The permission to assign supertype Ψ_x to *function term* x (on the right-hand side of $\vdash$) and still keep subtype Φ_x as type of *term variable* x in the associated term context (on the left-hand side of $\vdash$) yields a certain amount of (unparameterized) polymorphism due to subtyping. However, this sort of polymorphism is not very strong in its capabilities since we only allow restricted proved subtypes there, and we have seen in Proposition 4.55 that this amounts only to a change of occurrences of exponentials. But this is exactly, what subtyping was introduced for by Selinger and Valiron in simply typed *QLC*.

What happens when we remove the restriction on subtypes in the third premise and also allow the use of (*Top* supertype) for the derivation of this proved subtype? Here is an example:

$$\frac{\dfrac{\dfrac{\vdash x{:}!(\mathit{Top} \multimap \mathit{Bit})}{\vdash (\mathit{Top} \multimap \mathit{Bit})} \qquad \dfrac{\vdash x{:}!(\mathit{Top} \multimap \mathit{Bit})}{\vdash \mathit{Top}}}{\dfrac{\vdash\ !(\mathit{Top} \multimap \mathit{Bit}) <: (\mathit{Top} \multimap \mathit{Bit})}{x{:}!(\mathit{Top} \multimap \mathit{Bit}) \vdash x : (\mathit{Top} \multimap \mathit{Bit})}\ \text{(term variable)} \qquad \dfrac{\vdash\ !(\mathit{Top} \multimap \mathit{Bit}) <: \mathit{Top}}{x{:}!(\mathit{Top} \multimap \mathit{Bit}) \vdash x : \mathit{Top}}\ \text{(term variable)}}\ \text{(term application)}}{x{:}!(\mathit{Top} \multimap \mathit{Bit}) \vdash (x\ x) : \mathit{Bit}}$$

Obviously this leads to a stronger form of unparameterized polymorphism that even allows self application in certain places. But this is not what we aimed at with our polymorphic extension. Thus, we will not further investigate this in the present work and keep our rules in their restricted form.

However, we can still get near to the polymorphism in the above example, but in a more disciplined way by means of parameterized polymorphism, as the next example derivation shows:

$$\frac{\vdash x{:}!(\forall Y{<:}\mathit{Top}.Y) \qquad \begin{array}{c} \vdash (\forall Y{<:}\mathit{Top}.Y) \\ \vdash\ !(\forall Y{<:}\mathit{Top}.Y) \prec: (\forall Y{<:}\mathit{Top}.Y) \end{array}}{x{:}!(\forall Y{<:}\mathit{Top}.Y) \vdash x : (\forall Y{<:}\mathit{Top}.Y)}\ \text{(term variable)}\ .$$

We reuse this result twice as a starting point in derivation

$$\frac{\dfrac{\dfrac{x{:}!(\forall Y{<:}\mathit{Top}.Y) \vdash x : (\forall Y{<:}\mathit{Top}.Y)}{\vdash (\mathit{Top} \multimap \mathit{Bit})} \qquad \dfrac{x{:}!(\forall Y{<:}\mathit{Top}.Y) \vdash x : (\forall Y{<:}\mathit{Top}.Y)}{\vdash \mathit{Top}}}{\dfrac{\vdash (\mathit{Top} \multimap \mathit{Bit}) <: \mathit{Top}}{x{:}!(\forall Y{<:}\mathit{Top}.Y) \vdash (x\ (\mathit{Top} \multimap \mathit{Bit})) : (\mathit{Top} \multimap \mathit{Bit})}\ \text{(tyApp)} \qquad \dfrac{\vdash \mathit{Top} <: \mathit{Top}}{x{:}!(\forall Y{<:}\mathit{Top}.Y) \vdash (x\ \mathit{Top}) : \mathit{Top}}\ \text{(tyApp)}}\ \text{(tmApp)}}{x{:}!(\forall Y{<:}\mathit{Top}.Y) \vdash (x\ (\mathit{Top} \multimap \mathit{Bit})\ (x\ \mathit{Top})) : \mathit{Bit}}\ .$$

The result clearly is a parameterized variant of the former example.

There is one more interesting fact about rule (term variable), which concerns term context Γ. So far, we have met type bounds in the role of upper bounds (with respect to $<:$) in type contexts. In contrast to these, type terms in term contexts can be interpreted as *lower bounds* on the type of a term variable x. This is so because the actual essence of subtyping (and the sort of polymorphism it induces) is that each function term of type Φ is also of type Ψ as long as Ψ is a supertype of Φ.

Next, consider rules (linear term abstraction) and (nonlinear term abstraction). As the names already promise, the former rule yields a term abstraction of linear type and the latter results in a term abstraction of duplicable type. Besides this very obvious difference there are technical differences as well. We have already discussed in the sections on type terms that when we decide whether the type of $(\lambda x{:}\Phi_x.t)$ may be nonlinear, it does not matter whether the function represented by $(\lambda x{:}\Phi_x.t)$ takes an input of linear or nonlinear type Φ_x. And the same holds for the type Ψ of the output. What indeed does matter for this decision, is whether t contains free term variables of linear type. In this case, we would violate our linearity constraint as soon as $(\lambda x{:}\Phi_x.t)$ occurs more than once in a function term. However, we could allow free term variables of nonlinear type to appear in t, even if $(\lambda x{:}\Phi_x.t)$ is assigned a duplicable type $!^{n+1}(\Phi_x \multimap \Psi)$ and appears more than once in a function term. Hence, in rule (nonlinear term abstraction) we make sure there are no free term variables of linear type in t, before term abstraction $(\lambda x{:}\Phi_x.t)$ is assigned a duplicable type $!^{n+1}(\Phi_x \multimap \Psi)$. This check is facilitated by two mechanisms: firstly, we separate the involved term context into a nonlinear part $!\Delta$ and a part Γ that may contain both sorts of term variables – ones that are assigned a linear type, and ones that are assigned a nonlinear type (such a separation can be found in other derivation rules as well). Secondly, we ensure none of the term variables in the domain of Γ appear freely in function term t. This entails only term variables in the domain of $!\Delta$ may occur as free term variables in $(\lambda x{:}\Phi_x.t)$, and for these we definitely know they are assigned a duplicable type in $!\Delta$. (Please note Φ_x may be linear, though.) As we will see later (in Lemma 4.67), all free term variables must be in the domain of term context $!\Delta, \Gamma$, and thus we can be sure linearity is preserved if $(\lambda x{:}\Phi_x.t)$ appears more than once in a function term.

Now we come to derivation rule (term application). Here we employ the same pattern of separating term contexts into a nonlinear part $!\Delta$ and unrestricted parts Γ_1 and Γ_2. The crucial point is that the two premises share only the nonlinear part $!\Delta$. In the conclusion we have term context $!\Delta, \Gamma_1, \Gamma_2$, and according to the definition of terms contexts we know that domains $|!\Delta|$, $|\Gamma_1|$ and $|\Gamma_2|$ must be disjoint whenever we write $!\Delta, \Gamma_1, \Gamma_2$. Consequently, there can be no free term variable of linear type that appears in *s and t at the same time*. Hence, using this separation of term contexts guarantees linearity for term applications $(s\ t)$, and it is therefore of central importance. We will encounter this mechanism again in rules (pair term), (case distinction) and (recursion).

Derivation rules (linear type abstraction) and (nonlinear type abstraction) show a strong similarity to rules (linear-polymorphic type) and (nonlinear-polymorphic type), and as those complement each other, their respective counterparts on the function term level do as well. What is new, however, is the premise $X \notin \mathit{ftyv}(\Gamma)$ appearing in both rules. This is a necessary restriction that ensures type variable X is not used outside the scope Ψ of type abstraction $(\Lambda X{<:}\Phi_X.\Psi)$. Girard comments on this as follows:

> "[...] if we could form $\Lambda X.x^X$, what would then be the type of the free variable x in this expression? On the other hand, we can form $\Lambda X.\lambda x^X.x^X$ of type $\Pi X.X \to X$, which is the identity of any type."[47] ([GTL90], page 82)

When we pick up the first quoted sentence, we can make Girard's objection crystal clear when we ask a slightly modified question: what would the type of term variable x be in term-in-context $\vdash (\lambda x{:}X.(\Lambda X{<:}\mathit{Top}.x))$?

The last rule in Definition 4.63 formally introduces a concept on the level of type terms that we have been referring to on several occasions, namely the concept of type application. Although we already know that this mechanism is connected with substitution of free type variables, we now have it formalized. At last, we see in the derivation rule how upper bounds on type variables in type abstractions are guaranteed, namely by premise $\Theta \vdash \Upsilon <: \Phi_X$ (where premise $\Theta \vdash \Upsilon$ ensures that Υ is actually well-formed, as we have discussed earlier). Note, however, we did not use a restricted proved subtype in this premise. This is important indeed, since we intend to allow very general type bounds on type variables, such as *Top* (in

[47] In the quotation from Girard's well-known textbook a notation quite different from ours is used, and thus we transcribe it here: Girard's $\Lambda X.x^X$ corresponds to $x{:}X \vdash (\Lambda X{<:}\mathit{Top}.x) : \ldots$ in our notation, and "$\Lambda X.\lambda x^X.x^X$ of type $\Pi X.X \to X$" corresponds to term-in-context $\vdash (\Lambda X{<:}\mathit{Top}.(\lambda x{:}X.x)) : (\forall X{<:}\mathit{Top}.(X \multimap X))$ in our system.

order to allow any well-formed type term to be applied) or $!\mathit{Top}$ (to accept duplicable type terms only). We have already seen this mechanism at work in the above example derivations where we illustrated the necessity of restricting subtype derivations in rule (term variable).

In the next definition we introduce derivation rules for function terms related to pairs and tuples.

Definition 4.64 (proved terms – tuple rules).

$$\frac{\Theta \vdash \Gamma}{\Theta|\Gamma \vdash \langle\rangle : !^n \mathit{Unit}} \text{ (}\mathit{Unit}\text{ term)}$$

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash t_1 : !^n\Phi \qquad \Theta|!\Delta, \Gamma_2 \vdash t_2 : !^n\Psi}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash \langle t_1, t_2\rangle : !^n(\Phi \otimes \Psi)} \text{ (pair term)}$$

$$\frac{\Theta|\Gamma, x{:}!^n\Phi_x, y{:}!^n\Phi_y \vdash t : \Psi}{\Theta|\Gamma \vdash (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t) : (!^n(\Phi_x \otimes \Phi_y) \multimap \Psi)} \text{ (linear pair abstraction)}$$

$$\frac{\Theta|!\Delta, \Gamma, x{:}!^n\Phi_x, y{:}!^n\Phi_y \vdash t : \Psi \qquad \mathit{ftmv}(t) \cap |\Gamma| = \emptyset}{\Theta|!\Delta, \Gamma \vdash (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t) : !^m(!^n(\Phi_x \otimes \Phi_y) \multimap \Psi)} \text{ (nonlinear pair abstraction)}$$

The just defined derivation rules combine different mechanisms that we already encountered. On the one hand, we again see separation of term contexts into nonlinear and unrestricted parts, and also the reassurance that no term variables of linear type occur freely in the scope of a pair abstraction of duplicable type. Both mechanisms have already appeared in one of the earlier derivation rules for proved terms. On the other hand, we find that leading exponentials are propagated as it is done in rule (product type) on the isolated type level.

Interestingly, however, we do not need a dedicated derivation rule "(pair application)" for term applications of the form $((\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t)\ \langle t_1, t_2\rangle)$, since it can be perfectly covered by rule (term application) as follows:

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t) : !^m(!^n(\Phi_x \otimes \Phi_y) \multimap \Psi) \qquad \Theta|!\Delta, \Gamma_2 \vdash \langle t_1, t_2\rangle : !^n(\Phi_x \otimes \Phi_y)}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash \big((\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t)\ \langle t_1, t_2\rangle\big) : \Psi} \text{ (term application)}\ .$$

We have said earlier rule (*Unit* term) can be counted as axiom. But unlike the other axiom rules (new operator), (meas operator), (unitary operator) and (term variable), we do not have a restricted proved subtype as premise in (*Unit* type). This would indeed be an inconsistency if we did not use *restricted* proved types as premises in the other axiom rules. But we have defined rule (*Unit* term) so that it is equivalent to the following hypothetical derivation rule

$$\frac{\Theta \vdash \Gamma \qquad \begin{array}{c}\Theta \vdash \Psi \\ \Theta \vdash !\mathit{Unit} \prec: \Psi\end{array}}{\Theta|\Gamma \vdash \langle\rangle : \Psi}$$

which is undoubtedly analogous to the other axiom rules. And in this hypothetical rule we could very well remove the restriction on the proved subtype in the third premise if we desired to.

Let us finally define the last couple of derivation rules dealing with disjoint unions, case distinction and recursion terms.

Definition 4.65 (proved terms – case distinction and recursion).

$$\frac{\Theta|\Gamma \vdash t : !^n\Phi_l \qquad \Theta \vdash !^n\Phi_r}{\Theta|\Gamma \vdash \mathit{inj}_l(t) : !^n(\Phi_l \oplus \Phi_r)} \text{ (left injection)}$$

$$\frac{\Theta \vdash !^n\Phi_l \qquad \Theta|\Gamma \vdash t : !^n\Phi_r}{\Theta|\Gamma \vdash \mathit{inj}_r(t) : !^n(\Phi_l \oplus \Phi_r)} \text{ (right injection)}$$

$$\frac{\Theta|!\Delta,\Gamma_1 \vdash s : !^n(\Phi_x \oplus \Phi_y) \qquad \begin{array}{c}\Theta|!\Delta,\Gamma_2 \vdash (\lambda x{:}\Phi_x.t_l) : !^m(!^n\Phi_x \multimap \Psi)\\ \Theta|!\Delta,\Gamma_2 \vdash (\lambda y{:}\Phi_y.t_r) : !^m(!^n\Phi_y \multimap \Psi)\end{array}}{\Theta|!\Delta,\Gamma_1,\Gamma_2 \vdash \big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big) : \Psi}\ \text{(case distinction)}$$

$$\frac{\Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi) \vdash (\lambda x{:}\Phi_x.s) : !(\Phi_x \multimap \Psi) \qquad \Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi),\Gamma \vdash t : \Upsilon}{\Theta|!\Delta,\Gamma \vdash \big(letrec\ f{:}!(\Phi_x \multimap \Psi) = (\lambda x{:}\Phi_x.s)\ in\ t\big) : \Upsilon}\ \text{(recursion)}$$

These derivation rules do not come along with entirely new concepts or mechanisms.

However, rule (case distinction) constitutes an exception compared to what we have seen up to this point, because it softens our linearity constraint. The crucial point is that a term variable z of linear type may occur more than once in $\big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big)$, namely in function term t_l and *at the same time* in t_r. The reason for this is the shared term context Γ_2 in premises two and three. Please note that this is the only derivation rule which may introduce multiple occurrences of term variables having a linear type. Intuitively speaking, this is in accordance to our understanding of linearity, since the idea is that function terms of linear type shall contribute at most once to the overall computation. And the evaluation of case distinctions $(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda x{:}\Phi_y.t_r))$ ensures that only one of the two alternatives $((\lambda x{:}\Phi_x.t_l)\ v)$ (in case $s \equiv inj_l(v)$) or $((\lambda y{:}\Phi_y.t_r)\ w)$ (in case $s \equiv inj_r(w)$) is evaluated and never both. We will come back to this topic in subsection 4.6.3.

But there is one more noteworthy fact. In rule (recursion) term variable f (that stands as a placeholder for a term or pair abstraction) must be of duplicable type. The reason for this requirement is quite simple and intuitive: although f might occur at most once in t and also in s, it may nevertheless be reused multiple times (or even arbitrarily often) namely as soon as we perform the first truly recursive evaluation step in the course of evaluation.[48] For the same reason, no free term variables of linear type are allowed in $(\lambda x{:}\Phi_x.s)$, and therefore all types in the term context of the first premise are required to be duplicable.

After this in-depth discussion of our derivation rules for proved terms that distinguish the well-formed function terms from the rest, let us come to the consequences following from these rules. As in the section on type terms, we establish a result concerning the consistency of type and term contexts which are part of a proved term.

Lemma 4.66. *If $\Theta|\Gamma \vdash t : \Phi$ is a proved term, then*

(i) Θ is consistent,

(ii) Γ is consistent with respect to Θ, i.e. $\Theta \vdash \Gamma$ holds, and

(iii) $\Theta \vdash \Phi$ is a proved type.

Proof. We show each of these by induction on the derivation of proved term $\Theta|\Gamma \vdash t : \Phi$.

(i): Base cases: Let $\Theta|\Gamma \vdash t : \Phi$ be derived by rules (new operator), (meas operator), (unitary operator), (term variable), (type application), (left injection) or (right injection). Each of these rule has a premise of the form $\Theta \vdash \Phi'$ for some $\Phi' \in \mathcal{T}_{type}$. Hence, we may conclude consistency of Θ from these premises by Lemma 4.18.

Suppose $\Xi,\Xi'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : (\forall X{<:}\Phi_X.\Psi')$ has been derived using rule (linear type abstraction). The first premise of this rule already requires $\vdash \Xi,\Xi'$ and thus we immediately have consistency Ξ,Ξ'.

Next assume $\Xi,\Xi'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : !^n(\forall X{<:}\Phi_X.\Psi')$ has been derived using rule (nonlinear type abstraction). The first premise of this rule requires proved subtype $\Xi,\Xi' \vdash \Phi_X{<:}!Top$. Thus, by Lemma 4.19, we immediately get consistency of Ξ,Ξ'.

Now suppose $\Theta|\Gamma \vdash \langle\rangle : !^n\,Unit$ has been derived using rule (*Unit* term). The premise of this rule requires consistency of Γ with respect to Θ. If Γ is the empty term context, then, by Definition 4.58, we know Θ is consistent. If $\Gamma = x{:}\Phi_x,\Gamma'$ for some term variable x, type term Φ_x and term context Γ', then consistency of $x{:}\Phi_x,\Gamma'$ with respect to Θ implies that $\Theta \vdash \Phi_x$ is a proved type, and thus we get consistency of Θ by Lemma 4.18.

[48] The only case where this requirement gets superfluous is when either t or s does not contain f and the other function term contains f at most once. But then the whole usage of a recursion term in this place were superfluous, because the same effect could be reached by a regular term abstraction.

Induction cases: Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the rules (linear term abstraction), (nonlinear term abstraction), (term application), (pair term), (linear pair abstraction), (nonlinear pair abstraction), (case distinction) or (recursion). In each of these rules, we have at least one premise of the form $\Theta|\Gamma' \vdash t' : \Phi'$. We then get consistency of Θ by induction.

$\Diamond$

(ii): Base cases: Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the rules (new operator), (meas operator), (unitary operator), (term variable), (*Unit* term). Since we can find a premise $\Theta \vdash \Gamma$ in all of these rules, we immediately know term context Γ is consistent with respect to Θ.

Induction cases:

Case 1: rules (linear term abstraction), (nonlinear term abstraction), (linear pair abstraction), (nonlinear pair abstraction) and (recursion). In all these cases $\Theta|\Gamma \vdash t : \Phi$ has been derived from at least one premise of the form $\Theta|\Gamma, \Gamma' \vdash s : \Psi$. By induction we thus know $\Theta \vdash \Gamma, \Gamma'$ holds. Then, by definition of consistency of Γ, Γ' with respect to Θ (cf. Definition 4.58), we immediately know that both $\Theta \vdash \Gamma$ and $\Theta \vdash \Gamma'$ must hold.

Case 2: rules (term application), (pair term), (case distinction). Here Γ is of the form $!\Delta, \Gamma_1, \Gamma_2$. All of these rules have at least two premises of the form $\Theta|!\Delta, \Gamma_1 \vdash s_1 : \Psi_1$ and $\Theta|!\Delta, \Gamma_2 \vdash s_2 : \Psi_2$. By induction we thus get $\Theta \vdash !\Delta, \Gamma_1$ and $\Theta \vdash !\Delta, \Gamma_2$. But then, $\Theta \vdash !\Delta, \Gamma_1, \Gamma_2 = \Theta \vdash \Gamma$ also holds, due to definition of consistency of $!\Delta, \Gamma_1, \Gamma_2$ with respect to Θ.

Case 3: rules (linear type abstraction), (nonlinear type abstraction). Here Θ is of the form Ξ, Ξ'. Both of these rules contain a premise of the form $\Xi, X{<:}\Phi_X, \Xi'|\Gamma \vdash s : \Psi$. Let Γ be of the form $\Gamma = x_1{:}\Phi_1, \ldots, x_n{:}\Phi_n$ for some $n \geq 0$. Another premise in both rules assumes $X \notin \mathit{ftyv}(\Gamma)$, and thus we have $\mathit{ftyv}(\Phi_1), \ldots, \mathit{ftyv}(\Phi_n) \subseteq |\Xi, \Xi'|$. Moreover, on the one hand, consistency of Ξ, Ξ' is directly assumed by the first premise of rule (linear type abstraction), and, on the other hand, it is a consequence of Lemma 4.19 applied to the first premise of rule (nonlinear type abstraction). Hence, Ξ, Ξ' is also well-scoped due to Corollary 4.21. By induction, we get $\Xi, X{<:}\Phi_X, \Xi' \vdash \Gamma$, i.e. each of the $\Xi, X{<:}\Phi_X, \Xi' \vdash \Phi_i$ with $1 \leq i \leq n$ is a proved type. Hence, we may apply Corollary 4.28(i) to each of these, to obtain proved type $\Xi, \Xi' \vdash \Phi_i$, i.e. $\Theta \vdash \Gamma$.

Case 4: rules (type application), (left injection), (right injection). In all three rules, there is one premise of the form $\Theta|\Gamma \vdash s : \Psi$. From this, we immediately get $\Theta \vdash \Gamma$ by induction.

$\Diamond$

(iii): Base cases: Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the rules (new operator), (meas operator), (unitary operator), (term variable). For all of these cases we trivially get that $\Theta \vdash \Phi$ is a proved type, since there is a premise of the form $\Theta \vdash \Phi$ in each rule.

Assume $\Theta|\Gamma \vdash \langle\rangle : !^n \mathit{Unit}$ has been derived using rule (*Unit* term). From premise $\Theta \vdash \Gamma$ of this rule and Lemma 4.18 we know Θ is consistent. Hence, we can derive $\Theta \vdash !^n \mathit{Unit}$ by

$$\frac{\vdash \Theta}{\Theta \vdash !^n \mathit{Unit}} \text{ (\textit{Unit} type)} .$$

Induction cases: Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma, x{:}\Phi_x \vdash t' : \Psi'}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')} \text{ (linear term abstraction)} .$$

From the single premise and (ii) we get proved type $\Theta \vdash \Phi_x$ on the one hand. On the other hand, we obtain proved type $\Theta \vdash \Psi'$ by induction. Then we can derive $\Theta \vdash (\Phi_x \multimap \Psi')$ using rule (function type). In a similar way we can derive $\Theta \vdash !^{n+1}(\Phi_x \multimap \Psi')$ in case of rule (nonlinear term abstraction).

Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash s' : !^n(\Phi' \multimap \Psi') \qquad \Theta|!\Delta, \Gamma_2 \vdash t' : \Phi'}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash (s'\ t') : \Psi'} \text{ (term application)} .$$

By induction we obtain proved type $\Theta \vdash !^n(\Phi' \multimap \Psi')$ from the first premise of this rule. Inspecting all derivation rules for proved types, we find that this must have been derived by rule (function type). Hence, there must have been proved type $\Theta \vdash \Psi'$ due to the second premise in (function type). A similar argument covers rule (case distinction).

Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi'|\Gamma \vdash t' : \Psi' \qquad \begin{array}{c} X \notin \mathit{nftyv}(\Psi') \\ X \notin \mathit{ftyv}(\Gamma) \end{array}}{\Xi, \Xi'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t') : (\forall X{<:}\Phi_X.\Psi')} \text{ (linear type abstraction) .}$$

From the second premise we get proved type $\Xi, X{<:}\Phi_X, \Xi' \vdash \Psi'$ by induction. Using this and the other premises together, we can perform derivation

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi' \vdash \Psi' \qquad X \notin \mathit{nftyv}(\Psi')}{\Xi, \Xi' \vdash (\forall X{<:}\Phi_X.\Psi')} \text{ (linear-polymorphic type) ,}$$

yielding proved type $\Theta \vdash (\forall X{<:}\Phi_X.\Psi')$. We can handle the case of derivation rule (nonlinear type abstraction) in a similar way.

Consider the case where $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma \vdash t' : !^n(\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c} \Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X \end{array}}{\Theta|\Gamma \vdash (t'\ \Upsilon) : !^n\Psi'[\Upsilon/X]} \text{ (type application) .}$$

By induction we get proved type $\Theta \vdash !^n(\forall X <: \Phi_X.\Psi')$ from the first premise. Using this and the other premises, we can apply rule (bounded type substitution) from Theorem 4.50 to obtain $\Theta \vdash !^n\Psi[\Upsilon/X]$ by derivation

$$\frac{\Theta \vdash !^n(\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c} \Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X \end{array}}{\Theta \vdash !^n\Psi'[\Upsilon/X]} \text{ (bounded type substitution) .}$$

Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash t_1 : !^n\Phi' \qquad \Theta|!\Delta, \Gamma_2 \vdash t_2 : !^n\Psi'}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash \langle t_1, t_2\rangle : !^n(\Phi' \otimes \Psi')} \text{ (pair term) .}$$

Thus, we get proved types $\Theta \vdash !^n\Phi'$ and $\Theta \vdash !^n\Psi'$ from the two premises by induction. But then we can perform the following derivation to obtain proved type $\Theta \vdash !^n(\Phi' \otimes \Psi')$:

$$\frac{\Theta \vdash !^n\Phi' \qquad \Theta \vdash !^n\Psi'}{\Theta \vdash !^n(\Phi' \otimes \Psi')} \text{ (product type) .}$$

We may argue along the same lines for rules (left injection) and (right injection), where we need to apply derivation rule (sum type) instead of the above used (product type) rule.

Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma, x{:}!^n\Phi_x, y{:}!^n\Phi_y \vdash t' : \Psi'}{\Theta|\Gamma \vdash (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t') : (!^n(\Phi_x \otimes \Phi_y) \multimap \Psi')} \text{ (linear pair abstraction) .}$$

From the single premise and (ii) we may conclude $\Theta \vdash !^n\Phi_x$ and $\Theta \vdash !^n\Phi_y$ are both proved types. Furthermore, we get proved type $\Theta \vdash \Psi'$ by induction. Having these, we may perform derivation

$$\frac{\dfrac{\Theta \vdash !^n\Phi_x \qquad \Theta \vdash !^n\Phi_y}{\Theta \vdash !^n(\Phi_x \otimes \Phi_y)}\text{ (product type)} \qquad \Theta \vdash \Psi'}{\Theta \vdash !^m(!^n(\Phi_x \otimes \Phi_y) \multimap \Psi')} \text{ (function type)}$$

to finally obtain proved type $\Theta \vdash !^0(!^n(\Phi_x \otimes \Phi_y) \multimap \Psi')$ with $m = 0$. An analogous argument covers the case of rule (nonlinear pair abstraction).

Finally suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi') \vdash (\lambda x{:}\Phi_x.s) : !(\Phi_x \multimap \Psi') \qquad \Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi'), \Gamma' \vdash t' : \Upsilon}{\Theta|!\Delta, \Gamma' \vdash \big(letrec\ f{:}!(\Phi_x \multimap \Psi') = (\lambda x{:}\Phi_x.s)\ in\ t'\big) : \Upsilon}\ \text{(recursion)}\ .$$

We immediately get proved type $\Theta \vdash \Upsilon$ from the second premise by induction.

□

Remark: There is one result missing in the above lemma. Namely that every type term Ψ occurring in function term t of a proved term $\Theta|\Gamma \vdash t : \Phi$ is also derivable as part of a proved type $\widehat{\Theta} \vdash \Psi$ with $\Theta \sqsubseteq \widehat{\Theta}$. But this is a direct consequence of (i) and (ii) in Lemma 4.66, since each occurrence of a type term Ψ in t is introduced in one of four different ways:

- as part of a term or pair abstraction by one of the derivation rules (linear term abstraction), (nonlinear term abstraction), (linear pair abstraction) or (nonlinear pair abstraction);
- as part of a recursion term by rule (recursion);
- as part of a type abstraction by derivation rule (linear type abstraction) or (nonlinear type abstraction),
- or it is introduced as part of a type application by derivation rule (type application),

and in each of these cases we find a premise in the respective derivation rule to which we can apply Lemma 4.66 (i) or (ii) (or neither of these, as in the fourth case) and possibly take a (type weakening) step to derive an appropriate proved type $\widehat{\Theta} \vdash \Psi$. We do not go into formal details here, since we will not refer to this result later. But it is nevertheless an interesting and encouraging consequence – encouraging in the sense that we have found an appropriate set of derivation rules for our purposes, facilitating enforceable linearity, for instance.

What we get from Lemma 4.66 is not only that our derivation rules for proved terms preserve consistency of the involved type and term contexts, but we now know for sure that all type terms that occur in a proved term $\Theta|\Gamma \vdash t : \Phi$ are well-formed. In particular, we made sure no type (sub)term in Θ, Γ, t and Φ does violate Theorem 4.39 from subsection 4.2.3, and thus we have a good basis to show later on that a well-formed function term s, i.e. one for which we can derive a proved term $\Theta|\Gamma \vdash s : \Phi$, evaluates in accordance with the no-cloning theorem.

The property we confirm next ensures every free term variable occurring in proved term $\Theta|\Gamma \vdash t : \Phi$ also appears in Γ's domain.

Lemma 4.67. *If $\Theta|\Gamma \vdash t : \Phi$ is a proved term, then $ftmv(t) \subseteq |\Gamma|$.*

Proof. By induction of the derivation of $\Theta|\Gamma \vdash t : \Phi$:

Base cases: Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the rules (new operator), (meas operator), (unitary operator) or (*Unit* term). In these cases $ftmv(t) = \emptyset$ and we thus trivially have $ftmv(t) \subseteq |\Gamma|$.

Suppose $\Theta|\Gamma', x{:}\Phi_x \vdash x : \Psi_x$ has been derived by rule (term variable). Then it clearly holds $ftmv(x) = \{x\} \subseteq |\Gamma', x{:}\Phi_x| = |\Gamma'| \cup \{x\}$.

Induction cases: Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma, x{:}\Phi_x \vdash t' : \Psi'}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')}\ \text{(linear term abstraction)}\ .$$

By induction we then get $ftmv(t') \subseteq |\Gamma| \cup \{x\}$. Hence, we have

$$ftmv(\lambda x{:}\Phi_x.t') = ftmv(t') \setminus \{x\} \subseteq \big(|\Gamma| \cup \{x\}\big) \setminus \{x\} = |\Gamma|\ .$$

An analogous argument covers the cases of rules (nonlinear term abstraction), (linear pair abstraction) and (nonlinear pair abstraction).

Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta,\Gamma_1 \vdash s' : !^n(\Phi' \multimap \Psi') \qquad \Theta|!\Delta,\Gamma_2 \vdash t' : \Phi'}{\Theta|!\Delta,\Gamma_1,\Gamma_2 \vdash (s'\ t') : \Psi'}\ \text{(term application)}\ .$$

Induction then yields $ftmv(s') \subseteq |!\Delta,\Gamma_1|$ and $ftmv(t') \subseteq |!\Delta,\Gamma_2|$. From this we get

$$ftmv((s'\ t')) = ftmv(s') \cup ftmv(t') \subseteq |!\Delta,\Gamma_1| \cup |!\Delta,\Gamma_2| = |!\Delta,\Gamma_1,\Gamma_2|\ .$$

The same line of argument applies to the cases of rules (pair term) and (case distinction).

Now suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma \vdash t' : !^n(\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c}\Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array}}{\Theta|\Gamma \vdash (t'\ \Upsilon) : !^n\Psi'[\Upsilon/X]}\ \text{(type application)}\ .$$

Obviously, we have one premise from which we get $ftmv(t') \subseteq |\Gamma|$ by induction. Hence, we get

$$ftmv((t'\ \Upsilon)) = ftmv(t') \subseteq \Gamma\ .$$

We can cover the cases of derivation rules (linear type abstraction), (nonlinear type abstraction), (left injection) and (right injection) in an analogous fashion.

Finally, consider the case where $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi') \vdash (\lambda x{:}\Phi_x.s) : !(\Phi_x \multimap \Psi') \qquad \Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi'),\Gamma' \vdash t' : \Upsilon}{\Theta|!\Delta,\Gamma' \vdash (letrec\ f{:}!(\Phi_x \multimap \Psi') = (\lambda x{:}\Phi_x.s)\ in\ t') : \Upsilon}\ \text{(recursion)}\ .$$

Then we get by induction $ftmv((\lambda x{:}\Phi_x.s)) \subseteq |!\Delta| \cup \{f\}$ and $ftmv(t') \subseteq |!\Delta,\Gamma'| \cup \{f\}$. This yields

$$\begin{aligned} ftmv\big((letrec\ f{:}!(\Phi_x \multimap \Psi') = (\lambda x{:}\Phi_x.s)\ in\ t')\big) &= \big(ftmv((\lambda x{:}\Phi_x.s)) \cup ftmv(t')\big) \setminus \{f\} \\ &\subseteq \big(|!\Delta| \cup \{f\} \cup |!\Delta,\Gamma'| \cup \{f\}\big) \setminus \{f\} \\ &= |!\Delta,\Gamma'|\ . \end{aligned}$$

□

From this lemma we immediately know each of the term variables x that occur freely in a function term t in a proved term $\Theta|\Gamma \vdash t : \Phi$ has an assigned type term Φ_x (which can actually be derived as part of a proved type $\Theta \vdash \Phi_x$ due to Lemma 4.66).

At this point we are familiar with the most basic characteristics of proved terms. These fundamental technical insights will help us in getting along with the more advanced topics we deal with later.

4.4.2 Derived rules for proving terms

During our considerations on proved types we have developed a handy toolbox of derived derivation rules that supported us in dealing with more complex arguments. Two of the most valuable rules can be faithfully extended to the case of proved terms, namely weakening and permutation of type contexts.

The first derived rule we establish goes back to (type weakening). In addition to the weakening of the involved type context we also weaken the appearing term context in the same rule. The motivation for this joint weakening is that newly introduced variable-type pairs in the term context may require a weakening of the associated type context. We illustrate this with an example:

Suppose we are given proved terms

$$X{<:}Top|f{:}(X \multimap Bit) \vdash (\lambda x{:}X.f\ x) : (X \multimap Bit)$$

and

$$\vdash (\lambda y{:}Unit.inj_r(y)) : (Unit \multimap Bit)$$

and our goal is to derive proved term

$$X{<:}\mathit{Top}|f{:}(X \multimap \mathit{Bit}) \vdash (\mathit{match}\ s\ \mathit{with}\ (\lambda x{:}X.f\ x) \mid (\lambda y{:}\mathit{Unit}.\mathit{inj}_r(y))) : \mathit{Bit}\ .$$

Thus, before we can apply rule (case distinction), we have to weaken the second proved term so that its term context also contains variable-type pair $f{:}(X \multimap \mathit{Bit})$. But since this comes along with a free type variable, we also have to weaken the type context of the second proved term appropriately, so that the overall weakening eventually results in proved term

$$X{<:}\mathit{Top}|f{:}(X \multimap \mathit{Bit}) \vdash (\lambda y{:}\mathit{Unit}.\mathit{inj}_r(y)) : (\mathit{Unit} \multimap \mathit{Bit})\ .$$

Having this, we may then perform the following derivation to reach our goal:

$$\frac{X{<:}\mathit{Top}|\emptyset \vdash s : (X \oplus \mathit{Unit}) \qquad \begin{array}{c} X{<:}\mathit{Top}|f{:}(X \multimap \mathit{Bit}) \vdash (\lambda x{:}X.f\ x) : (X \multimap \mathit{Bit}) \\ X{<:}\mathit{Top}|f{:}(X \multimap \mathit{Bit}) \vdash (\lambda y{:}\mathit{Unit}.\mathit{inj}_r(y)) : (\mathit{Unit} \multimap \mathit{Bit}) \end{array}}{X{<:}\mathit{Top}|f{:}(X \multimap \mathit{Bit}) \vdash (\mathit{match}\ s\ \mathit{with}\ (\lambda x{:}X.f\ x) \mid (\lambda y{:}\mathit{Unit}.\mathit{inj}_r(y))) : \mathit{Bit}}\ (\text{caseDist})\ .$$

The just motivated joint weakening rule for type and term contexts has the following form.

Proposition 4.68. *We can derive derivation rule*

$$\frac{\widehat{\Theta} \vdash \widehat{\Gamma} \qquad \Theta|\Gamma \vdash t : \Phi \qquad \begin{array}{c} \Theta \sqsubseteq \widehat{\Theta} \\ \Gamma \subseteq \widehat{\Gamma} \end{array}}{\widehat{\Theta}|\widehat{\Gamma} \vdash t : \Phi}\ \textit{(term weakening)}\ . \qquad (\star)$$

Proof. We show this by induction on the derivation of proved term $\Theta|\Gamma \vdash t : \Phi$. But before we start, we first take a look at premise $\widehat{\Theta} \vdash \widehat{\Gamma}$. This stands for a proved type $\widehat{\Theta} \vdash \widehat{\Gamma}(x)$ for each $x \in |\widehat{\Gamma}|$. Hence, we know that Θ is consistent by Lemma 4.18. Thus, we can assume another premise stating $\vdash \Theta$. This together with premise $\Theta \sqsubseteq \widehat{\Theta}$ will enable the application of rules (type weakening) and (subtype weakening) from subsection 4.2.2 in several of the following cases.

Base cases: Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta \vdash \Gamma \qquad \begin{array}{c} \Theta \vdash \Psi \\ \Theta \vdash !(\mathit{Bit} \multimap \mathit{Qbit}) \prec: \Psi \end{array}}{\Theta|\Gamma \vdash \mathrm{new} : \Psi}\ (\text{new operator})\ .$$

Using (type weakening) and (subtype weakening), we then obtain proved type $\widehat{\Theta} \vdash \Psi$ and restricted proved subtype $\widehat{\Theta} \vdash !(\mathit{Bit} \multimap \mathit{Qbit}) \prec: \Psi$ from the second and third premise. And since we already have $\widehat{\Theta} \vdash \widehat{\Gamma}$ as a premise in $(\star)$, we may perform derivation

$$\frac{\widehat{\Theta} \vdash \widehat{\Gamma} \qquad \begin{array}{c} \widehat{\Theta} \vdash \Psi \\ \widehat{\Theta} \vdash !(\mathit{Bit} \multimap \mathit{Qbit}) \prec: \Psi \end{array}}{\widehat{\Theta}|\widehat{\Gamma} \vdash \mathrm{new} : \Psi}\ (\text{new operator})\ .$$

Similar reasoning applies in the cases of rules (meas operator), (unitary operator), (term variable) and (Unit term).

Induction cases: Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma, x{:}\Phi_x \vdash t' : \Psi'}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')}\ (\text{linear term abstraction})\ ,$$

where we assume $x \notin |\widehat{\Gamma}|$ (without loss of generality, since we identify α-equivalent function terms). Then we get proved term $\widehat{\Theta}|\widehat{\Gamma}, x{:}\Phi_x \vdash t' : \Psi'$ by induction. From this we can immediately derive $\widehat{\Theta}|\widehat{\Gamma} \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')$ by the same derivation rule (linear term abstraction). Analogous reasoning applies to the case of rule (linear pair abstraction).

Now assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta,\Gamma',x{:}\Phi_x \vdash t' : \Psi' \qquad \mathit{ftmv}(t') \cap |\Gamma'| = \emptyset}{\Theta|!\Delta,\Gamma' \vdash !^{n+1}(\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')}\ \text{(nonlinear term abstraction)}\ ,$$

where $!\Delta,\Gamma' = \Gamma$ and where we again assume $x \notin |\widehat{\Gamma}|$. From premise $\mathit{ftmv}(t') \cap |\Gamma'| = \emptyset$ and the fact $\mathit{ftmv}(t') \subseteq |!\Delta,\Gamma',x{:}\Phi_x|$, which is due to Lemma 4.67, we then know all free term variables in t' already appear in term context $(!\Delta,x{:}\Phi_x)$, i.e. $\mathit{ftmv}(t') \subseteq |!\Delta,x{:}\Phi_x|$ and even $\mathit{ftmv}(t') \setminus \{x\} \subseteq |!\Delta|$. Since $!\Delta \subseteq (!\Delta,\Gamma') \subseteq \widehat{\Gamma}$ and since x does not appear in $|\widehat{\Gamma}|$ by assumption, we clearly have $\mathit{ftmv}(t') \cap \big|\widehat{\Gamma} \setminus !\Delta\big| = \emptyset$. Moreover, we get proved term $\widehat{\Theta}|\widehat{\Gamma},x{:}\Phi_x \vdash t' : \Psi'$ by induction, which we may rewrite into $\widehat{\Theta}|!\Delta,\big(\widehat{\Gamma} \setminus !\Delta\big),x{:}\Phi_x \vdash t' : \Psi'$. Having this, we may perform derivation

$$\frac{\widehat{\Theta}|!\Delta,\big(\widehat{\Gamma} \setminus !\Delta\big),x{:}\Phi_x \vdash t' : \Psi' \qquad \mathit{ftmv}(t') \cap \big(\widehat{\Gamma} \setminus !\Delta\big) = \emptyset}{\widehat{\Theta}|\underbrace{!\Delta,\big(\widehat{\Gamma} \setminus !\Delta\big)}_{=\,\widehat{\Gamma}} \vdash !^{n+1}(\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')}\ \text{(nonlinear term abstraction)}\ .$$

We can cover the case of rule (nonlinear pair abstraction) by similar arguments.

Next suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta,\Gamma_1 \vdash s' : !^n(\Phi' \multimap \Psi') \qquad \Theta|!\Delta,\Gamma_2 \vdash t' : \Phi'}{\Theta|!\Delta,\Gamma_1,\Gamma_2 \vdash (s'\ t') : \Psi'}\ \text{(term application)}\ ,$$

where $!\Delta,\Gamma_1,\Gamma_2 = \Gamma$. We here apply induction to both premises, but using slightly differing extended term contexts.
Firstly, we inductively apply rule (term weakening) to the first premise using extended type context $\widehat{\Theta}$ with $\Theta \sqsubseteq \widehat{\Theta}$ and extended term context $\widehat{\Gamma} \setminus \Gamma_2$ with $\big(!\Delta\Gamma_1\big) \subseteq \big(\widehat{\Gamma} \setminus \Gamma_2\big)$. This yields proved term $\widehat{\Theta}|\widehat{\Gamma} \setminus \Gamma_2 \vdash s' : !^n(\Phi' \multimap \Psi')$, where we can rewrite term context $\widehat{\Gamma} \setminus \Gamma_2$ to $\big(!\Delta,\big(\big(\widehat{\Gamma} \setminus \Gamma_2\big) \setminus !\Delta\big)\big)$.
Secondly, we inductively apply (term weakening) to the second premise, still with extended type context $\widehat{\Theta}$, but this time with "extended" term context $!\Delta,\Gamma_2$ (which surely is a subset of itself). In this way we obtain proved term $\widehat{\Theta}|!\Delta,\Gamma_2 \vdash t' : \Phi'$. We can now perform derivation

$$\frac{\widehat{\Theta}|!\Delta,\big(\big(\widehat{\Gamma} \setminus \Gamma_2\big) \setminus !\Delta\big) \vdash s' : !^n(\Phi' \multimap \Psi') \qquad \widehat{\Theta}|!\Delta,\Gamma_2 \vdash t' : \Phi'}{\widehat{\Theta}|\underbrace{!\Delta,\big(\big(\widehat{\Gamma} \setminus \Gamma_2\big) \setminus !\Delta\big),\Gamma_2}_{=\,\widehat{\Gamma}} \vdash (s'\ t') : \Psi'}\ \text{(term application)}$$

to derive the proved term we desire. Clearly, the cases where $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the rules (pair term), (case distinction) and (recursion) can be handled in a similar way.

Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\vdash \Xi,\Xi' \qquad \Xi,X{<:}\Phi_X,\Xi'|\Gamma \vdash t' : \Psi' \qquad \begin{array}{c} X \notin \mathit{nftyv}(\Psi') \\ X \notin \mathit{ftyv}(\Gamma) \end{array}}{\Xi,\Xi'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t') : (\forall X{<:}\Phi_X.\Psi')}\ \text{(linear type abstraction)}\ ,$$

where $\Xi,\Xi' = \Theta$, and where we assume $X \notin |\widehat{\Theta}|$ and $X \notin \mathit{ftyv}(\widehat{\Gamma})$ (without loss of generality, since we identify α-equivalent proved terms).
Let $\widehat{\Xi}$ and $\widehat{\Xi}'$ be two term contexts such that $\Xi \sqsubseteq \widehat{\Xi}$ and $\Xi' \sqsubseteq \widehat{\Xi}'$ and $\widehat{\Xi},\widehat{\Xi}' = \widehat{\Theta}$. By consistency of $\widehat{\Theta} = \widehat{\Xi},\widehat{\Xi}'$ we also know $\widehat{\Xi}$ is consistent. By consistency of $\Xi,X{<:}\Phi_X,\Xi'$ (which we get from the second premise by Lemma 4.66 (i)) we know $\Xi \vdash \Phi_X$ is a proved type. Using rule (type weakening) we obtain proved type $\widehat{\Xi} \vdash \Phi_X$ from this, which yields consistency of $\widehat{\Xi},X{<:}\Phi_X$. Let $\widehat{\Xi}'$ be of the form $\widehat{\Xi}' = Y_1{<:}\Phi'_1,\ldots,Y_k{<:}\Phi'_k$ for some $k \geq 0$. Then we have for each i with $1 \leq i \leq k$ a proved type

$$\widehat{\Xi},Y_1{<:}\Phi'_1,\ldots,Y_{i-1}{<:}\Phi'_{i-1} \vdash \Phi'_i$$

by consistency of $\widehat{\Xi}, \widehat{\Xi}'$. Now we iteratively apply (type weakening) again for each $i = 1, \ldots, k$ to obtain the following proved types (and the implied consistency results) in a step-by-step fashion

$$\begin{aligned} \widehat{\Xi}, X{<:}\Phi_X &\vdash \Phi'_1 \ , \\ \widehat{\Xi}, X{<:}\Phi_X, Y_1{<:}\Phi'_1 &\vdash \Phi'_2 \ , \\ &\vdots \\ \widehat{\Xi}, X{<:}\Phi_X, Y_1{<:}\Phi'_1, \ldots, Y_{k-1}{<:}\Phi'_{k-1} &\vdash \Phi'_k \ , \end{aligned}$$

which eventually leads to consistency of $\widehat{\Xi}, X{<:}\Phi_X, \widehat{\Xi}'$.
Having this together with the fact $\big(\Xi, X{<:}\Phi_X, \Xi'\big) \sqsubseteq \big(\widehat{\Xi}, X{<:}\Phi_X, \widehat{\Xi}'\big)$, we can now apply induction to premise $\Xi, X{<:}\Phi_X, \Xi' | \Gamma \vdash t' : \Psi'$ and thus obtain proved term $\widehat{\Xi}, X{<:}\Phi_X, \widehat{\Xi}' | \widehat{\Gamma} \vdash t' : \Psi'$. And since we assume $X \notin \mathit{ftyv}(\widehat{\Gamma})$, we can now perform derivation

$$\frac{\vdash \widehat{\Xi}, \widehat{\Xi}' \qquad \widehat{\Xi}, X{<:}\Phi_X, \widehat{\Xi}' | \widehat{\Gamma} \vdash t' : \Psi' \qquad \begin{array}{c} X \notin \mathit{nftyv}(\Psi') \\ X \notin \mathit{ftyv}(\widehat{\Gamma}) \end{array}}{\underbrace{\widehat{\Xi}, \widehat{\Xi}'}_{=\,\widehat{\Theta}} | \widehat{\Gamma} \vdash (\Lambda X{<:}\Phi_X.t') : (\forall X{<:}\Phi_X.\Psi')} \text{ (linear type abstraction)} \ .$$

We may argue similarly for the case of rule (nonlinear type abstraction). However, there we additionally have to derive $\widehat{\Theta} \vdash \Phi_X <: !\mathit{Top}$ by (subtype weakening) from premise $\Theta \vdash \Phi_X <: !\mathit{Top}$.

Next assume $\Theta | \Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta | \Gamma \vdash t' : !^n(\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c} \Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X \end{array}}{\Theta | \Gamma \vdash (t'\ \Upsilon) : !^n \Psi'[\Upsilon/X]} \text{ (type application)} \ .$$

Then we use induction to get $\widehat{\Theta} | \widehat{\Gamma} \vdash t' : !^n(\forall X <: \Phi_X.\Psi')$, and we apply (type weakening) and (subtype weakening) to obtain $\widehat{\Theta} \vdash \Upsilon$ and $\widehat{\Theta} \vdash \Upsilon <: \Phi_X$, respectively. Using these as premises, we can perform the derivation

$$\frac{\widehat{\Theta} | \widehat{\Gamma} \vdash t' : !^n(\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c} \widehat{\Theta} \vdash \Upsilon \\ \widehat{\Theta} \vdash \Upsilon <: \Phi_X \end{array}}{\widehat{\Theta} | \widehat{\Gamma} \vdash (t'\ \Upsilon) : !^n \Psi'[\Upsilon/X]} \text{ (type application)} \ .$$

This form of reasoning can be easily transferred to the cases of rules (left injection) and (right injection).

□

In contrast to rule (type weakening), we do not need to explicitly formulate a premise in (term weakening) requiring consistency of $\widehat{\Theta}$, since this is implicit in $\widehat{\Theta} \vdash \widehat{\Gamma}$, due to Lemma 4.18.

For an extension of (type permutation) to the level of proved terms, we do not have to bother with permutations of terms contexts, since these are defined as sets and thus do not show a specific order among the variable-type pairs they contain. Hence, the form of rule (term permutation) is almost the same as that of (type permutation).

Proposition 4.69. *We can extend rule (type permutation) from Proposition 4.25 to the setting of proved terms:*

$$\frac{\vdash \Theta^\pi \qquad \Theta | \Gamma \vdash t : \Phi}{\Theta^\pi | \Gamma \vdash t : \Phi} \textit{ (term permutation)} \ .$$

Proof. We prove this proposition by induction on the derivation of $\Theta | \Gamma \vdash t : \Phi$.

Base cases: Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta \vdash \Gamma \qquad \begin{array}{c}\Theta \vdash \Psi \\ \Theta \vdash !(\mathit{Bit} \multimap \mathit{Qbit}) \prec: \Psi\end{array}}{\Theta|\Gamma \vdash \text{new} : \Psi}\ \text{(new operator)}\ .$$

Since premise $\Theta \vdash \Gamma$ stands for a set of proved types $\Theta \vdash \Gamma(x)$ for all $x \in |\Gamma|$, we may apply (type permutation) to each of these and eventually obtain $\Theta^\pi \vdash \Gamma$. Moreover, we can apply (type permutation) and (subtype permutation) to the second and third premise to get proved type $\Theta^\pi \vdash \Psi$ and restricted proved subtype $\Theta^\pi \vdash !(\mathit{Bit} \multimap \mathit{Qbit}) \prec: \Psi$. Using these as premises, we may derive $\Theta^\pi|\Gamma \vdash \text{new} : \Psi$ by rule (new operator).
The same is true for rules (meas operator), (unitary operator), (term variable) and (Unit term).

Induction cases: Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma, x{:}\Phi_x \vdash t' : \Psi'}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')}\ \text{(linear term abstraction)}\ .$$

By induction we get proved term $\Theta^\pi|\Gamma, x{:}\Phi_x \vdash t' : \Psi'$ from the single premise. We may then apply rule (linear term abstraction) to this again to obtain proved term $\Theta^\pi|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')$, and we are done in this case.

The cases where $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the rules (nonlinear term abstraction), (term application), (type application), (pair term), (linear pair abstraction), (nonlinear pair abstraction), (left injection), (right injection), (case distinction) and (recursion) can all be handled by a combination of the reasoning for the two cases we have seen so far.

Now assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\vdash \Xi, \Xi' \qquad \Xi, X{<:}\Phi_X, \Xi'|\Gamma \vdash t' : \Psi' \qquad \begin{array}{c}X \notin \mathit{nftyv}(\Psi') \\ X \notin \mathit{ftyv}(\Gamma)\end{array}}{\Xi, \Xi'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t') : (\forall X{<:}\Phi_X.\Psi')}\ \text{(linear type abstraction)}\ .$$

where $\Xi, \Xi' = \Theta$. Before we can apply induction to the second premise, we need to make sure $\Theta^\pi, X{<:}\Phi_X$ is consistent. When we apply Lemma 4.66(i) to the second premise, we get consistency of $\Xi, X{<:}\Phi_X, \Xi'$, which entails $\Xi \vdash \Phi_X$ is a proved type. Since we already know Ξ, Ξ' is consistent due to the first premise, we may apply (type weakening) to obtain $\Xi, \Xi' \vdash \Phi_X$. And since we have $\Xi, \Xi' = \Theta$ and $\vdash \Theta^\pi$, we may now apply (type permutation) to get proved type $\Theta^\pi \vdash \Phi_X$, which entails consistency of $\Theta^\pi, X{<:}\Phi_X$. Hence, since this is a permuted and consistent version of type context $\Xi, X{<:}\Phi_X, \Xi'$, we can apply induction to the second premise to derive proved term $\Theta^\pi, X{<:}\Phi_X|\Gamma \vdash t' : \Psi'$. Thus, we have all necessary pieces together to perform derivation

$$\frac{\vdash \Theta^\pi \qquad \Theta^\pi, X{<:}\Phi_X|\Gamma \vdash t' : \Psi' \qquad \begin{array}{c}X \notin \mathit{nftyv}(\Psi') \\ X \notin \mathit{ftyv}(\Gamma)\end{array}}{\Theta^\pi|\Gamma \vdash (\Lambda X{<:}\Phi_X.t') : (\forall X{<:}\Phi_X.\Psi')}\ \text{(linear type abstraction)}\ .$$

The case of rule (nonlinear type abstraction) can be handled similarly. However, there we get consistency of Ξ, Ξ' by application of Lemma 4.19 to premise $\Xi, \Xi' \vdash \Phi_X <: !\mathit{Top}$ and we need to use rule (subtype permutation) to obtain proved subtype $\Theta^\pi \vdash \Phi_X <: !\mathit{Top}$ from the same premise. □

The next derived rule amounts to a strengthening of term contexts in proved terms. We had a similar result for type contexts in Lemma 4.27. However, the conditions in the following lemma are less complex due to the simpler nature of term contexts in comparison to type contexts.

Lemma 4.70 (adapted and generalized from Lemma 1.3.24(1) in [SV09])**.**
Let $\Theta|\Gamma, \Gamma' \vdash t : \Phi$ be a proved term. If $x \notin \mathit{ftmv}(t)$ for each term variable $x \in |\Gamma'|$, then we can also derive proved term $\Theta|\Gamma \vdash t : \Phi$.

Proof. We show this by induction on the derivation of $\Theta|\Gamma,\Gamma' \vdash t : \Phi$.

Base cases: Assume $\Theta|\Gamma,\Gamma' \vdash \mathit{new} : \Psi$ has been derived using rule (new operator). The first premise then states $\Theta \vdash \Gamma,\Gamma'$. From this consistency result, we conclude also Γ alone is consistent with respect to Θ. And since the other premises do not mention Γ or Γ', we thus can derive $\Theta|\Gamma \vdash \mathrm{new} : \Psi$ using the same rule. Of course, the same argument holds for rules (meas operator), (unitary operator) and (*Unit* term).

Suppose $\Theta|\Gamma,\Gamma' \vdash x : \Psi_x$ has been derived by rule (term variable). Then by the first and third premises of this rule, variable-type pair $x{:}\Phi_x$ must appear in Γ for some type term Φ_x (such that $\Theta \vdash \Phi_x \prec: \Psi_x$ is a restricted proved subtype), since $x \in \mathit{ftmv}(x)$ and we assume $|\Gamma'|$ does not contain any term variables that occur freely in t. As above, we get from this consistency result also the consistency statement $\Theta \vdash \Gamma$, and thus we may derive $\Theta|\Gamma \vdash x : \Psi_x$.

Induction cases: Suppose $\Theta|\Gamma,\Gamma' \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma,\Gamma',x{:}\Phi_x \vdash t' : \Psi'}{\Theta|\Gamma,\Gamma' \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')}\ \text{(linear term abstraction)}\ .$$

Because of assumption $y \notin |\Gamma'|$ for all $y \in \mathit{ftmv}((\lambda x{:}\Phi_x.t')) = \mathit{ftmv}(t') \setminus \{x\}$, and due to the fact $x \notin |\Gamma'|$ (since otherwise $\Gamma,\Gamma',x{:}\Phi_x$ would not be a proper term context), we know none of the free term variables in t' does occur in $|\Gamma'|$. Hence, we then get proved term $\Theta|\Gamma \vdash t' : \Psi'$ by induction. Consequently, we may derive proved term $\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')$ using the same rule. A similar line of argument can be applied in the cases of rules (nonlinear term abstraction), (linear pair abstraction) and (nonlinear pair abstraction).

Now suppose $\Theta|\Gamma,\Gamma' \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta,!\Delta',\Gamma_1,\Gamma_1' \vdash s' : !^n(\Phi' \multimap \Psi') \qquad \Theta|!\Delta,!\Delta',\Gamma_2,\Gamma_2' \vdash t' : \Phi'}{\Theta|!\Delta,!\Delta',\Gamma_1,\Gamma_1',\Gamma_2,\Gamma_2' \vdash (s'\ t') : \Psi'}\ \text{(term application)}\ ,$$

where $!\Delta,\Gamma_1,\Gamma_2 = \Gamma$ and $!\Delta',\Gamma_1',\Gamma_2' = \Gamma'$. Since none of the term variables in $\mathit{ftmv}((s'\ t')) = \mathit{ftmv}(s') \cup \mathit{ftmv}(t')$ appears in $|!\Delta',\Gamma_1',\Gamma_2'|$, it is clear that none of the term variables in $|!\Delta',\Gamma_1'|$ appears in $\mathit{ftmv}(s')$ and none of the term variables in $|!\Delta',\Gamma_2'|$ appears in $\mathit{ftmv}(t')$. Hence, we obtain proved terms $\Theta|!\Delta,\Gamma_1 \vdash s' : !^n(\Phi' \multimap \Psi')$ and $\Theta|!\Delta,\Gamma_2 \vdash t' : \Phi'$ by induction. Having these, we may then use the same rule to derive proved term $\Theta|!\Delta,\Gamma_1,\Gamma_2 \vdash (s'\ t') : \Psi'$.
We can handle the cases of derivation rules (pair term) and (case distinction) analogously.

Assume $\Theta|\Gamma,\Gamma' \vdash t : \Phi$ has been derived by one of the rules (linear type abstraction), (nonlinear type abstraction), (type application), (left inclusion), (right inclusion). In each of these rules there in one premise of the form $\Theta'|\Gamma,\Gamma' \vdash t' : \Psi'$, where function term t' is a subexpression of t in so that $\mathit{ftmv}(t) = \mathit{ftmv}(t')$. Hence, we can apply induction to this premise and thus obtain proved term $\Theta'|\Gamma \vdash t' : \Psi'$. Since none of the mentioned rules takes the included term context Γ further into account, we may thus apply the respective rule again to all premises using the modified one, however, and in this way derive proved term $\Theta|\Gamma \vdash t : \Phi$.

Finally, the case where $\Theta|\Gamma,\Gamma' \vdash t : \Phi$ has been derived by (recursion) can be covered by combing the arguments from cases (linear term abstraction) and (term application).

□

For the moment this already concludes our efforts to derive further rules for the derivation of proved terms. We will suggest the derivation of a few more rules in the section on type safety, more precisely in subsection 4.6.2.

Next we turn our attention to operational semantics for polymorphically typed *QLC* function terms.

4.5 Operational semantics

We have already presented operational semantics for untyped *QLC* in section 3.2. Furthermore, we have said we follow the Curry-style of language definition or, in other words, we consider semantics *prior* to typing. Consequently, our operational semantics for polymorphically typed *QLC* does not essentially differ from that of untyped *QLC*, although it is extended appropriately in order to deal with explicit typing annotations and to implement a mechanism concerned with the syntactic constructs of type abstraction and type application.

At first we modify our notion of fully evaluated function terms – called value terms – to the setting of polymorphically typed function terms. As we have term and pair abstractions among value terms in untyped *QLC*, we keep these as value terms, however, supplemented with explicit type annotations, but additionally we also consider type abstractions as fully evaluated. Hence, we end up with the following modified and extended definition.

Definition 4.71 (polymorphically typed value terms).
In analogy to Definition 3.7 we inductively define the set $\mathcal{T}_{pValue}$ of *polymorphically typed value terms* (or *value terms* for short when there is no danger of confusion) by

$$\begin{aligned} c, x, \langle\rangle &\in \mathcal{T}_{pValue} \text{ , where } c \in \mathcal{C}_{term} \text{ and } x \in \mathcal{V}_{term} \text{ ,} \\ (\lambda x{:}\Phi_x.t), (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t) &\in \mathcal{T}_{pValue} \text{ , where } x, y \in \mathcal{V}_{term},\ \Phi_x, \Phi_y \in \mathcal{T}_{type} \text{ and } t \in \mathcal{T}_{pQLC} \text{ ,} \\ inj_l(v), inj_r(v) &\in \mathcal{T}_{pValue} \text{ , where } v \in \mathcal{T}_{pValue} \text{ ,} \\ \langle v, w\rangle &\in \mathcal{T}_{pValue} \text{ , where } v, w \in \mathcal{T}_{pValue} \text{ ,} \\ (\Lambda X{<:}\Phi_X.t) &\in \mathcal{T}_{pValue} \text{ , where } X \in \mathcal{V}_{type},\ \Phi_X \in \mathcal{T}_{type} \text{ and } t \in \mathcal{T}_{pQLC} \text{ .} \end{aligned}$$

We moreover need to modify our definition of quantum closures. The main novelty is that we require polymorphically typed function terms embedded in quantum closures to not contain any free type variables. Thus, the name "closure" still keeps its intention since the linking sequence binds the remaining free term variables in the embedded function term t.

Definition 4.72 (quantum closures over polymorphically typed function terms).
A *quantum closure* over polymorphically typed function terms is a triple $[Q, L, t]$, where $Q \in \mathcal{H}_n$ and $L = |q_1, \ldots, q_n\rangle$ are defined as in the untyped setting, and $t \in \mathcal{T}_{pQLC}$ is a polymorphically typed function term which does not contain free type variables and whose free term variables all appear in L, i.e. $ftyv(t) = \emptyset$ and $ftmv(t) \subseteq \{q_1, \ldots, q_n\}$.

Next we transfer the notion of well-typedness to quantum closures. In the definition we have just given all remaining free term variables in t are bound by linking sequence $L = |q_1, \ldots, q_n\rangle$, and we have seen (when presenting operational semantics for untyped *QLC* in section 3.2) that each of these free term variables $q_1, \ldots, q_n$ is associated to exactly one qubit in state Q. Hence, we intuitively know each of the $q_1, \ldots, q_n$ shall be assigned type *Qbit*. We could explicitly annotate this in the linking sequence itself, for instance as $|q_1{:}Qbit, \ldots, q_n{:}Qbit\rangle$. But since this is quite uncomfortable and in fact does not carry any interesting information (since none of the $q_1, \ldots, q_n$ could possibly be assigned a different type than *Qbit*), we rather renounce explicit annotations of the type of term variables $q_1, \ldots, q_n$. Nevertheless, we implicitly use the discussed type assignments when we introduce the following concept of well-typedness for quantum closures.

Definition 4.73 (well-typed state).
A quantum closure $[Q, L, t] = [Q, |q_1, \ldots, q_n\rangle, t]$ is considered *well-typed*, if there exists a type term Φ such that

$$q_1{:}Qbit, \ldots, q_n{:}Qbit \vdash t : \Phi$$

is a proved term. We write $[Q, L, t] : \Phi$ to denote a well-typed quantum closure together with its associated type term. We then call $[Q, L, t] : \Phi$ a *well-typed state*.

In the following definition we present the basic reduction rules that implement the "classical control" part of polymorphically typed *QLC*. As we have done during our presentation of operational semantics for untyped *QLC*, we take a simplified point of view for these basic reduction rules and ignore (at least for the time being) the parts of quantum closures that remain unchanged by these rules. That means each of the below defined basic reduction rules $s \to t$ induces a probabilistic reduction rule $[Q, L, s] \to_1 [Q, L, t]$ on quantum closures.

Definition 4.74 (basic reduction rules for polymorphically typed function terms).
We syntactically extend the basic reduction rules we have introduced in Definition 3.8:

$$\begin{aligned} (\lambda x{:}\Phi_x.t)\ v &\to t[v/x] \text{ ,} \\ (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t)\ \langle v, w\rangle &\to t[v/x, w/y] \text{ ,} \end{aligned}$$

$$\begin{aligned} &\mathit{match}\ \mathit{inj}_l(v)\ \mathit{with}\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r) \to (\lambda x{:}\Phi_x.t_l)\ v\ , \\ &\mathit{match}\ \mathit{inj}_r(w)\ \mathit{with}\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r) \to (\lambda y{:}\Phi_y.t_r)\ w\ , \\ &\qquad \mathit{letrec}\ f{:}\Phi_f = (\lambda x{:}\Phi_x.s)\ \mathit{in}\ t \to t\big[(\lambda x{:}\Phi_x.(\mathit{letrec}\ f{:}\Phi_f = (\lambda x{:}\Phi_x.s)\ \mathit{in}\ s))\ /\ f\big]\ . \end{aligned}$$

In addition, we get one completely new basic reduction rule:

$$(\Lambda X{<:}\Phi_X.t)\ \Psi \to t[\Psi/X]\ .$$

At this point we have seen all non-trivial changes that are necessary to adapt operational semantics defined for untyped *QLC* to obtain operational semantics for polymorphically typed *QLC*. Hence, it is enough to say that we adapt the remaining Definitions 3.9, 3.10, 3.11 and 3.12 accordingly, so that the definition of operational semantics for our polymorphic extension of *QLC* is complete in the spirit of the definitions in the current section.

Let us now illustrate operational semantics, typing of function terms and the relationships of untyped *QLC*, simply typed *QLC* and polymorphically typed *QLC* in an example, for which the missing details are elaborated in appendix C.1. For brevity we concentrate on the involved function terms rather than full quantum closures, since their form is only of marginal interest for the subsequent considerations.

Example 4.75. Let us come back to the fair quantum coin we already encountered in Example 3.5 modeled by untyped *QLC* function term

$$\mathrm{coin} \equiv \big(\lambda z.\mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big)\ .$$

Recall that we can computationally model a toss of our quantum coin by term application $(\mathrm{coin}\ \langle\rangle)$ which then evaluates to a value term representing a random bit. Since the shape of the resulting value term is determined by chance (but lies in the range $\{0,1\}$, for which we have defined $0 \equiv \mathit{inj}_r(\langle\rangle)$ and $1 \equiv \mathit{inj}_l(\langle\rangle)$ in section 3), we denote the final result of a full evaluation of $(\mathrm{coin}\ \langle\rangle)$ by $(\mathrm{coin}\ \langle\rangle)^{\downarrow}$.

In the *untyped* setting, we may use Church numerals

$$\ulcorner n \urcorner \equiv \big(\lambda f.\big(\lambda x.(\underbrace{f\ (\ldots(f}_{n\ \text{times}}\ x)\ldots)\big)\big)$$

to formulate a function term whose full evaluation results in a tuple of n random bits in an iterative manner when it is evaluated according to *QLC*'s operational semantics. Thus, the following evaluation of function term

$$t_{\mathrm{toss}} :\equiv \big(\big(\ulcorner n-1 \urcorner\ (\lambda y.\langle \mathrm{coin}\ \langle\rangle,\ y\rangle)\big)\ (\mathrm{coin}\ \langle\rangle)\big)$$

may be interpreted as n tosses of our fair quantum coin:

$$\big(\ulcorner n-1 \urcorner\ (\lambda y.\langle \mathrm{coin}\ \langle\rangle,\ y\rangle)\big)\ (\mathrm{coin}\ \langle\rangle) \to^* \big\langle \underbrace{(\mathrm{coin}\ \langle\rangle)^{\downarrow}, \ldots, (\mathrm{coin}\ \langle\rangle)^{\downarrow}}_{n\ \text{times}} \big\rangle\ .$$

(See appendix C.1.2 for a more detailed presentation of this evaluation.)

In the *simply typed* setting we cannot derive a type for t_{toss}. The problem lies in the type of term variable f in function term $\ulcorner n-1 \urcorner$, since, on the one hand, each occurrence of f in subexpression $(f\ (\ldots(f\ x)\ldots))$ would need to be assigned the same type. On the other hand, function term $(\lambda y.\langle \mathrm{coin}\ \langle\rangle,\ y\rangle)$ needs to be assigned a type of the form $(\Phi \multimap (\Psi \otimes \Phi))$, where Φ is the type assigned to term variable y and Ψ is the type assigned to $(\mathrm{coin}\ \langle\rangle)$. Hence, in order to derive a simple (i.e. non-polymorphic) type for term variable f in $\ulcorner n-1 \urcorner$ in t_{toss}, we would need to unify type terms such as $(\Phi \multimap (\Psi \otimes \Phi))$ and $((\Psi \otimes \Phi) \multimap (\Psi \otimes (\Psi \otimes \Phi)))$. This clearly leads to a recursive structure and finally to an infinite type term, which is in this general form beyond the capabilities of simply typed *QLC*.[49]

[49] In recursive type systems one can deal with such recursive types by defining appropriate isomorphisms, see chapters 20 and 21 in [Pie02] or system $\lambda\mu$ in section 4 in [Bar92] for more details.

In the *polymorphically typed* setting, we can extend Church numerals by type abstractions and type applications to enable a polymorphic typing with respect to tuples, i.e. right-nested pairs:

$$\ulcorner n \urcorner^{!\otimes} :\equiv \big(\Lambda X{<:}!Top.\big(\lambda f{:}!\Phi_f.\big(\lambda x{:}!X.(f\ X^{!\otimes n}\ (f\ X^{!\otimes n-1}(\ldots(f\ !X\ x)\ldots)))\big)\big)\big)\big) \ ,$$

where Φ_f stands as abbreviation for type term

$$\Phi_f := (\forall Y{<:}!Top.(!Y \multimap !!(!X \otimes Y)))$$

and where we define the $\cdot^{!\otimes n}$ operator by

$$\begin{aligned} \Phi^{!\otimes 1} &:= !\Phi \ , \\ \Phi^{!\otimes m+1} &:= !(!\Phi \otimes \Phi^{!\otimes m}) \end{aligned}$$

for arbitrary $\Phi \in \mathcal{T}_{type}$ and for all $m \geq 0$. Furthermore, we annotate the type assigned to z in function term coin in the obvious way $\text{coin}_{\text{poly}} :\equiv \big(\lambda z{:}\,Unit.\text{meas}\ (H\ (\text{new}\ 0))\big)$, and having done this, we also lift function term t_{toss} to the polymorphic setting:

$$t_{\text{poly}} :\equiv \Big(\ulcorner n-1 \urcorner^{!\otimes}\ !Bit\ \big(\Lambda Y{<:}!Top.(\lambda y{:}!Y.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle)\big)\Big)\ \big(\text{coin}_{\text{poly}}\ \langle\rangle\big) \ .$$

In appendix C.1.1 we show how to derive proved terms $\vdash (\text{coin}_{\text{poly}}\ \langle\rangle) : !!Bit$ and also $\vdash t_{\text{poly}} : !(!Bit)^{!\otimes n}$. According to operational semantics for polymorphically typed *QLC* as we have defined it in the current subsection, we get the following full evaluation of function term t_{poly}:[50]

$$\begin{aligned} &\Big(\ulcorner n-1 \urcorner^{!\otimes}\ !Bit\ \big(\Lambda Y{<:}!Top.(\lambda y{:}!Y.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle)\big)\Big)\ \big(\text{coin}_{\text{poly}}\ \langle\rangle\big) \\ &\qquad \to^* \big\langle \underbrace{(\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}, \ldots, (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}}_{n \text{ times}} \big\rangle \ . \end{aligned}$$

Let us take a look at how we can derive a type for the final result of this evaluation. We have said that each occurrence of $(\text{coin}_{poly}\ \langle\rangle)^{\downarrow}$ either equals $1 \equiv inj_l(\langle\rangle)$ or $0 \equiv inj_r(\langle\rangle)$, respectively. We can perform the following derivations to obtain proved terms $\vdash 1 : !^n Bit$ and $\vdash 0 : !^n Bit$ for any $n \geq 0$. (Please recall, we have defined $Bit = (Unit \oplus Unit)$ in Convention 4.60.)

$$\dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash \langle\rangle : !^n Unit}\,(Unit \text{ term}) \qquad \dfrac{\vdash \emptyset}{\vdash !^n Unit}\,(Unit \text{ type})}{\underbrace{\vdash inj_l(\langle\rangle) : !^n(Unit \oplus Unit)}_{\equiv\, \vdash 1 : !^n Bit}}\,(\text{left injection}) \qquad \dfrac{\dfrac{\vdash \emptyset}{\vdash !^n Unit}\,(Unit \text{ type}) \qquad \dfrac{\vdash \emptyset}{\vdash \langle\rangle : !^n Unit}\,(Unit \text{ term})}{\underbrace{\vdash inj_r(\langle\rangle) : !^n(Unit \oplus Unit)}_{\equiv\, \vdash 0 : !^n Bit}}\,(\text{right injection})$$

Since we can assign the same type in both cases, this directly translates to proved term $\vdash (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow} : !^n Bit$ for any $n \geq 0$. We use this knowledge to perform the following derivation

$$\dfrac{\dfrac{\vdash (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow} : !^{n+2}Bit \qquad \vdash (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow} : !^{n+2}Bit}{\vdash \big\langle (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}, (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow} \big\rangle : !^n(!!Bit \otimes !!Bit)}\,(\text{pair term}) \qquad \vdots}{\vdash \big\langle \underbrace{(\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}, \ldots, (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}}_{n \text{ times}} \big\rangle : \underbrace{!!(!!Bit \otimes (!Bit)^{!\otimes n-1})}_{= !(!Bit)^{!\otimes n}}}\,(\text{pair term})$$

from which we eventually obtain proved term

$$\vdash \big\langle (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}, \ldots, (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow} \big\rangle : !(!Bit)^{!\otimes n} \ .$$

This result is another indication towards the type preservation property that we will investigate further in subsection 4.6.2.

[50] Again, for a more detailed presentation see appendix C.1.2.

The above example gives some insight into the expressiveness of polymorphically typed *QLC* in contrast to simply typed *QLC*. Although a detailed investigation of this expressiveness might be of interest, we do not follow this path in depth since it is clearly beyond the scope of the present work.

Nevertheless, we take one more quick glance at expressiveness. At the end of our considerations concerned with untyped *QLC* in section 3 we have shown there are function terms t in *QLC* for which a quantum closure $[Q, L, t]$ does not have a full evaluation or, in other words, we never reach an irreducible state when evaluating $[Q, L, t]$. We have also given an example for such a function term, namely

$$letrec\ f = (\lambda x.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)\ .$$

A natural and interesting question is: can we assign a type to this function term (equipped with appropriate type annotations)? The simple answer to this question is given by proved term

$$\vdash \big(letrec\ f{:}!(Unit \multimap \Psi) = (\lambda x{:}Unit.(f\ \langle\rangle))\ in\ (f\ \langle\rangle)\big) : \Psi\ ,$$

where we may even choose an arbitrary type term Ψ as long as $\vdash \Psi$ can be derived as a proved type. A derivation of the above proved term can be found in appendix C.2.

4.6 Type safety

In the beginning of section 4.4 we have stated that the purpose of deriving proved terms is to distinguish the subset of well-formed terms or, in other words, to rule out error terms (cf. Definition 4.56). But that is only one aspect of the important concept of type safety. On its own this property named *progress property* is not worth much. Imagine we have a well-typed state $[Q, |q_1, \ldots, q_l\rangle, t] : \Phi$. What we then know due to the progress property is that t is not an error term. That means, t either is a value term or t is reducible. But what can we say about $[Q', |q_1, \ldots, q_{l'}\rangle, t']$ that results from an evaluation step $[Q, |q_1, \ldots, q_l\rangle, t] \rightarrow_p [Q', |q_1, \ldots, q_{l'}\rangle, t']$? To make any statement about the well-formedness of t', we then have to try to find a type term Φ' and a derivation of proved term $q_1{:}Qbit, \ldots, q_{l'}{:}Qbit \vdash t' : \Phi'$ (at runtime!). However, in order to find out whether $[Q', L', t']$ is an error state or not, we could also just syntactically check whether t' is a value term and otherwise check all reduction rules for their applicability, instead of performing a presumably more expensive typability check. Hence, it becomes clear that we need another property that complements the progress property. The sought property is called *type preservation* and we already started to investigate our type system with respect to type preservation in subsection 4.2.5. However, for the sake of completeness, we here again give a compact description of what type preservation and the progress property are about: (adapted from [Pie02], page 95)

Progress: A well-typed quantum closure either is a value state or it can take an evaluation step according to operational semantics, i.e. it is not an error state.

Type preservation: If a well-typed quantum closure takes an evaluation step, then the resulting quantum closure is also well-typed.

The description of the type preservation property is somewhat weaker than the name actually indicates, since when a well-typed quantum closure $[Q, L, t]$ of type Φ is evaluated by $[Q, L, t] \rightarrow_p [Q', L', t']$, then type preservation, as it is described above, says there is a type term Ψ such that $[Q', L', t'] : \Psi$ is a well-typed state. However, type preservation does not necessarily imply $\Phi = \Psi$ although the name indicates it. Indeed, many type systems guarantee this strong version of type preservation, but there are some that do not. In subsection 4.6.2 we take a look at which form of type preservation our system (presumably) supplies.

Obviously, it is desirable to have both the progress property and the type preservation property together, since then it is enough to show for a quantum closure $[Q, L, t]$ that it is well-typed to ensure that all evaluation results $[Q'', L'', t'']$ with $[Q, L, t] \rightarrow_p^* [Q'', L'', t'']$ are well-typed as well. In other words, we then know that an evaluation of $[Q, L, t]$ cannot arrive at an evaluation result $[Q'', L'', t'']$ that is neither reducible nor a value state.

Due to the limited scope of the present work, we will not develop the full formal details that are necessary to prove the type preservation property of polymorphically typed *QLC*. Nevertheless, we sketch a possible path that may lead to this result in subsection 4.6.2 and we have already pointed out some positive indications towards type preservation throughout the preceding sections. Regarding the progress property, we conduct further investigations in subsection 4.6.1 which eventually lead to a confirmation of the validity of this property. These investigations put together yield the following conjectured result:

Conjecture 4.76 (type safety, adapted from Corollary 1.3.33 in [SV09])**.**
A well-typed quantum closure in polymorphically typed QLC does not reach an error state during evaluation. In other words, if there exists a full evaluation for a quantum closure $[Q, L, t]$ with final result $[Q^{\downarrow}, L^{\downarrow}, t^{\downarrow}]$, then $t^{\downarrow}$ is a value term.

As already said, this would be a direct consequence of the progress property in combination with type preservation (in its strong form or in the weak variant that we stated above).

Remark: Type preservation in its strong form does indeed hold for simply typed *QLC*. This is confirmed by Theorem 1.3.30 in [SV09]. However, Selinger and Valiron call this property by its alternative (also widely used) name "*subject reduction*." On the other hand, also the progress property does hold for simply typed *QLC* (as it is shown by Theorem 1.3.32 in [SV09]).

In addition to the considerations on "classical" type safety, we show in subsection 4.6.3 that for each well-typed state $[Q, |q_1, \ldots, q_l\rangle, t] : \Phi$ function term t fulfills a certain linearity constraint which entails, roughly speaking, that none of the $q_1, \ldots, q_l$ appears more than once freely in t. This is true except for the case where a q_j with $1 \leq j \leq l$ appears at the same time in alternatives $(\lambda x{:}\Phi_x.t_l)$ and $(\lambda y{:}\Phi_y.t_r)$ in a case distinction term $(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r))$. Of course, this property is also preserved during evaluation if the type preservation property holds for our polymorphic extension of *QLC*.

4.6.1 Progress property

We start by showing that polymorphically typed *QLC* exhibits the progress property. In a preliminary step we establish a helpful lemma telling us what the possible forms of a value term v and the associated type term Φ in a proved term $\Theta|\Gamma \vdash v : \Phi$ can be.

Lemma 4.77 (canonical forms[51], adapted and extended from Lemma 1.3.31 in [SV09])**.**
Let $\Theta|\Gamma \vdash v : \Phi$ be a proved term where v is a polymorphically typed value term. Then exactly one of the following cases applies (m and n may be equal to zero or greater in each case):

(i) $v \in \mathcal{C}_{term}$ *is a term constant,*

(ii) $v \in \mathcal{V}_{term}$ *is a term variable,*

(iii) $v = \langle\rangle$ *and* Φ *is of the form* $\Phi = !^n\, Unit$*,*

(iv) v *is of the form* $v = (\lambda x{:}\Phi_x.t)$ *and* Φ *is of the form* $\Phi = !^n(\Phi_x \multimap \Psi')$*,*

(v) v *is of the form* $v = (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t)$ *and* Φ *is of the form* $\Phi = !^m(!^n(\Phi_x \otimes \Phi_y) \multimap \Psi')$*,*

(vi) v *is of the form* $v = inj_l(w)$ *or* $v = inj_r(w)$ *and* Φ *is of the form* $\Phi = !^n(\Phi' \oplus \Psi')$ *in both cases,*

(vii) v *is of the form* $v = \langle w_1, w_2\rangle$ *and* Φ *is of the form* $\Phi = !^n(\Phi' \otimes \Psi')$*, or*

(viii) v *is of the form* $v = (\Lambda X{<:}\Phi_X.t)$ *and* Φ *is of the form* $\Phi = !^n(\forall X{<:}\Phi_X.\Psi')$*.*

Proof. We confirm this by inspection of the derivation rules for proved terms. What we find is that each of the cases (i) to (viii) is associated to one or more derivation rules according to the following pattern.

(i): (new operator), (meas operator) and (unitary operator)
(ii): (term variable)
(iii): (*Unit* term)
(iv): (linear term abstraction) and (nonlinear term abstraction)
(v): (linear pair abstraction) and (nonlinear pair abstraction)
(vi): (left injection) and (right injection)
(vii): (pair term)
(viii): (linear type abstraction) and (nonlinear type abstraction)

[51] The name "canonical forms" has been taken over from [Pie02].

That means, for instance, if $\Theta|\Gamma \vdash v : \Phi$ has been derived by one of the rules (new operator), (meas operator) or (unitary operator), then case (i) applies and no other case. This is due to the fact that none of the above mentioned rules is associated to two distinct cases.

On the other hand, whenever a proved term $\Theta|\Gamma \vdash t : \Phi$ has been derived by one of the derivation rules (term application), (type application), (case distinction) or (recursion), then we know t cannot be a value term. This becomes evident when we inspect these derivation rules and the definition of value terms.

□

By now we have not explicitly formulated the result of Lemma 4.77, but we have already been using it implicitly in several places. In these places, we referred to an "inspection of the derivation rules" or we have been using similar formulations. Now that we have explicitly stated the connection between the forms of function terms and type terms in proved terms and the derivation rule the respective proved term has been derived with, we can use it in the proof of the progress property that is established in the following theorem.

Theorem 4.78 (progress)**.** *Let $[Q_0, L_0, t_0] : \Psi$ be a well-typed state with $L_0 = |q_1, \ldots, q_k\rangle$ for some $k \geq 0$, i.e. $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ is a proved term. Then either t_0 is a value term or there exists a quantum closure $[Q_1, L_1, t_1]$ such that we have a reduction step $[Q_0, L_0, t_0] \to_p [Q_1, L_1, t_1]$ for some real number $p \in [0, 1]$ from the unit interval.*

Proof. We prove this by induction on the derivation of proved term $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$:

Base cases: Suppose $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived by one of the rules (new operator), (meas operator), (unitary operator), (term variable), (linear term abstraction), (nonlinear term abstraction), (linear type abstraction), (nonlinear type abstraction), (*Unit* term), (linear pair abstraction) or (nonlinear pair abstraction). By inspection of all these rules, we find that t_0 must then be a value term, i.e. $t_0 \in \mathcal{T}_{pValue}$.

Induction case: Assume $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived by

$$\frac{\Gamma_1 \vdash s' : !^m(\Phi \multimap \Psi) \qquad \Gamma_2 \vdash t' : \Phi}{q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash (s'\ t') : \Psi}\ \text{(term application)}$$

for some $m \geq 0$ and where we set $\Gamma_1, \Gamma_2 := q_1{:}Qbit, \ldots, q_k{:}Qbit$. We may use (term weakening) to obtain proved terms $\Gamma_1, \Gamma_2 \vdash s' : !^m(\Phi \multimap \Psi)$ and $\Gamma_1, \Gamma_2 \vdash t' : \Phi$, and thus $[Q_0, L_0, s'] : !^m(\Phi \multimap \Psi)$ and $[Q_0, L_0, t'] : \Phi$ are well-typed states. Then, by induction, we need to distinguish several cases:

$[Q_0, L_0, t']$ is reducible and there thus is an evaluation step $[Q_0, L_0, t'] \to_p [Q_1, L_1, t'']$. In this case we may employ the following instance of a congruence rule from Definition 3.11 (lifted to the polymorphic setting)

$$\frac{[Q_0, L_0, t'] \to_p [Q_1, L_1, t'']}{[Q_0, L_0, s'\ t'] \to_p [Q_1, L_1, s'\ t'']}$$

and thus come up with evaluation step $[Q_0, L_0, s'\ t'] \to_p [Q_1, L_1, s'\ t'']$.

t' is a value term and $[Q_0, L_0, s']$ is reducible with $[Q_0, L_0, s'] \to_p [Q_1, L_1, s'']$ being an adequate evaluation step. Then the following instance of a congruence rule helps to get an evaluation step $[Q_0, L_0, s'\ t'] \to_p [Q_1, L_1, s''\ t']$:

$$\frac{[Q_0, L_0, s'] \to_p [Q_1, L_1, s'']}{[Q_0, L_0, s'\ t'] \to_p [Q_1, L_1, s''\ t']}\ .$$

s' and t' are value terms. By Lemma 4.77 and since every free term variable in t_0 is assigned type *Qbit* (due to Lemma 4.67 and term context $q_1{:}Qbit, \ldots, q_k{:}Qbit$) we conclude s' can only take on one of the following forms:

$s' =$ new. In this case we conclude (by derivation rule (new operator) and Proposition 4.55) that $!^m(\Phi \multimap \Psi)$ is of the form $!^m(!^n Bit \multimap Qbit)$. We can thus collect the following facts:

- t' is of type $!^n Bit = !^n(Unit \oplus Unit)$, cf. Convention 4.60,

- t' is a value term,
- term constants are always assigned function types,
- all free term variables in t' have type $Qbit$ (due to Lemma 4.67).

Hence, by Lemma 4.77, t' must be of the form $t' = inj_l(\langle\rangle) \equiv 1$ or $t' = inj_r(\langle\rangle) \equiv 0$. But then we can perform one of the two reduction steps

$$\left[Q_0, |q_1, \ldots, q_k\rangle, \text{new } 1\right] \to_1 \left[Q_0 \otimes |1\rangle, |q_1, \ldots, q_k, q_{k+1}\rangle, q_{k+1}\right]$$

or

$$\left[Q_0, |q_1, \ldots, q_k\rangle, \text{new } 0\right] \to_1 \left[Q_0 \otimes |0\rangle, |q_1, \ldots, q_k, q_{k+1}\rangle, q_{k+1}\right],$$

respectively, according to Definition 3.10 (lifted to the polymorphic setting).

$s' = \text{meas}$. Then we conclude (by derivation rule (meas operator), Proposition 4.55 and Theorem 4.39) that $!^m(\Phi \multimap \Psi)$ is of the form $!^m(Qbit \multimap !^n Bit)$.[52] In analogy to the above induction case we can collect the following facts:

- t' is of type $Qbit$,
- t' is a value term,
- term constants are always assigned function types,
- all free term variables in t' have type $Qbit$ (due to Lemma 4.67).

It thus follows t' must be a (free) term variable in range $\{q_1, \ldots, q_k\}$, i.e. $t' = q_i$ with $1 \leq i \leq k$. But then we can perform one of the two reduction steps

$$\left[\alpha Q_{0,0} + \beta Q_{0,1}, |q_1, \ldots, q_k\rangle, \text{meas } q_i\right] \to_{|\alpha|^2} \left[Q_{0,0}, |q_1, \ldots, q_k\rangle, 0\right]$$

or

$$\left[\alpha Q_{0,0} + \beta Q_{0,1}, |q_1, \ldots, q_k\rangle, \text{meas } q_i\right] \to_{|\beta|^2} \left[Q_{0,1}, |q_1, \ldots, q_k\rangle, 1\right],$$

respectively, also according to the lifted variant of Definition 3.10.

$s' = (\lambda x{:}\Phi_x.s'')$. This case is fairly easy since we can immediately apply evaluation step $\left[Q_0, L_0, (\lambda x{:}\Phi_x.s'')\ t'\right] \to_1 \left[Q_0, L_0, s''[t'/x]\right]$ according to Definition 4.74 and the lifted variant of Definition 3.10.

$s' = (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.s'')$. Due to Lemma 4.77 we conclude Φ in $!^m(\Phi \multimap \Psi)$ has the form $\Phi = !^n(\Phi_x \otimes \Phi_y)$. Since all term constants are assigned a function type and since all free term variables in t' are of type $Qbit$, Lemma 4.77 also tells us t' of type $!^n(\Phi_x \otimes \Phi_y)$ must be of the form $t' = \langle t'_1, t'_2\rangle$. And since we assume t' is a value term, t'_1 and t'_2 are both value terms. Hence, the following evaluation step is possible

$$\left[Q_0, L_0, (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.s'')\ \langle t'_1, t'_2\rangle\right] \to_1 \left[Q_0, L_0, s''[t'_1/x, t'_2/y]\right].$$

Suppose $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived by

$$\frac{q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : (\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c}\vdash \Upsilon \\ \vdash \Upsilon <: \Phi_X\end{array}}{q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash (t_0\ \Upsilon) : \Psi'[\Upsilon/X]}\ (\text{type application}).$$

Since all term constants are assigned function types, and since all free terms variables are of type $Qbit$, we conclude by Lemma 4.77 that t_0 is of the form $t_0 = (\Lambda X{<:}\Phi_X.t')$. Due to Definition 4.74 we then immediately have evaluation step

$$\left[Q_0, L_0, (\Lambda X{<:}\Phi_X.t')\ \Upsilon\right] \to_1 \left[Q_0, L_0, t'[\Upsilon/X]\right].$$

[52] In this argument Theorem 4.39 tells us there is no leading exponential in front of $Qbit$ because of premise $\vdash \Psi$ in rule (meas operator).

Assume $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived by

$$\frac{\Gamma_1 \vdash t_1' : !^n\Psi_1' \qquad \Gamma_2 \vdash t_2' : !^n\Psi_2'}{q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash \langle t_1', t_2'\rangle : !^n(\Psi_1' \otimes \Psi_2')}\ \text{(pair term)}\ ,$$

where we set $\Gamma_1, \Gamma_2 := q_1{:}Qbit, \ldots, q_k{:}Qbit$. In analogy to the induction step of rule (term application), we find well-typed states $[Q_0, L_0, t_1'] : !^n\Psi_1'$ and $[Q_0, L_0, t_2'] : !^n\Psi_2'$ using (term weakening). Hence, by induction, we need to distinguish three cases:

- $[Q_0, L_0, t_2']$ is reducible by an evaluation step $[Q_0, L_0, t_2'] \to_p [Q_1, L_1, t_2'']$. Then, using the appropriate congruence rule from the lifted variant of Definition 3.11, we come to evaluation step $\big[Q_0, L_0, \langle t_1', t_2'\rangle\big] \to_p \big[Q_1, L_1, \langle t_1', t_2''\rangle\big]$.
- t_2' is a value term and $[Q_0, L_0, t_1']$ is reducible by an evaluation step $[Q_0, L_0, t_1'] \to_p [Q_1, L_1, t_1'']$. Symmetrically to the previous case, we then get evaluation step $\big[Q_0, L_0, \langle t_1', t_2'\rangle\big] \to_p \big[Q_1, L_1, \langle t_1'', t_2'\rangle\big]$, using the appropriate congruence rule.
- t_1' and t_2' are both value terms. Consequently, $\langle t_1', t_2'\rangle$ is also a value term.

We may argue along similar (even simpler) lines for the cases where $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived using rules (left injection) or (right injection).

Suppose $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived by

$$\frac{\begin{array}{cc} & \Gamma_2 \vdash (\lambda x{:}\Phi_x.t_l) : !^m(!^n\Phi_x \multimap \Psi) \\ \Gamma_1 \vdash s : !^n(\Phi_x \oplus \Phi_y) & \Gamma_2 \vdash (\lambda y{:}\Phi_y.t_r) : !^m(!^n\Phi_y \multimap \Psi)\end{array}}{q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash \big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big) : \Psi}\ \text{(case distinction)}\ ,$$

where we set $\Gamma_1, \Gamma_2 := q_1{:}Qbit, \ldots, q_k{:}Qbit$. Using rule (term weakening), we get proved term $\Gamma_1, \Gamma_2 \vdash s : !^n(\Phi_x \oplus \Phi_y)$ from the first premise, and thus we have well-typed state $[Q_0, L_0, s] : !^n(\Phi_x \oplus \Phi_y)$. By induction, we then need to distinguish two cases:

- $[Q_0, L_0, s]$ is reducible and we have evaluation step $[Q_0, L_0, s] \to_p [Q_1, L_1, s'']$. Then we can use the following instance of a congruence rule from Definition 3.11 to obtain the desired evaluation step $[Q_0, L_0, t_0] \to_p [Q_1, L_1, t_1]$:

$$\frac{\big[Q_0, L_0, s\big] \to_p \big[Q_1, L_1, s''\big]}{\begin{array}{l}\big[Q_0, L_0, match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big] \\ \qquad \to_p \big[Q_1, L_1, match\ s''\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big]\end{array}}\ .$$

- s is a value term. By Lemma 4.77 and due to the facts that all term constants are assigned a function type and all free term variables in s have type $Qbit$, we conclude s is of the form $inj_l(s')$ or $inj_r(s')$, where $s' \in \mathcal{T}_{pValue}$ is a value term in both cases. Depending on the exact form of s, Definition 4.74 tells us which of the following two reduction steps can be performed:

$$\big[Q_0, L_0, match\ inj_l(s')\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big] \to_1 \big[Q_0, L_0, (\lambda x{:}\Phi_x.t_l)\ s'\big]$$

or

$$\big[Q_0, L_0, match\ inj_r(s')\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big] \to_1 \big[Q_0, L_0, (\lambda y{:}\Phi_y.t_r)\ s'\big]\ .$$

Finally suppose $q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t_0 : \Psi$ has been derived by

$$\frac{f{:}!(\Phi_x \multimap \Upsilon) \vdash (\lambda x{:}\Phi_x.s) : !(\Phi_x \multimap \Upsilon) \qquad f{:}!(\Phi_x \multimap \Upsilon), q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash t' : \Psi}{q_1{:}Qbit, \ldots, q_k{:}Qbit \vdash (letrec\ f{:}!(\Phi_x \multimap \Upsilon) = (\lambda x{:}\Phi_x.s)\ in\ t') : \Psi}\ \text{(recursion)}\ .$$

According to Definition 4.74 and the lifted variant of Definition 3.10, we can perform evaluation step

$$\Big[Q_0, L_0, \mathit{letrec}\ f{:}!(\Phi_x \multimap \Upsilon) = (\lambda x{:}\Phi_x.s)\ \mathit{in}\ t'\Big]$$
$$\to_1 \Big[Q_0, L_0, t'\big[(\lambda x{:}\Phi_x.(\mathit{letrec}\ f{:}!(\Phi_x \multimap \Upsilon) = (\lambda x{:}\Phi_x.s)\ \mathit{in}\ s))\ /\ f\big]\Big]\ .$$

□

This result clearly shows that we cannot derive a proved term $\Theta|\Gamma \vdash e : \Phi$ for any error term e.

Remark: It is worthwhile to compare the just proven result to its counterpart Theorem 1.3.32 in [SV09] which is concerned with the progress property for simply typed *QLC*. There are two interesting respects in which Selinger's and Valiron's result is stronger than ours.

Firstly, in Theorem 1.3.32 in [SV09] it is not only stated that if a well-typed quantum closure $[Q_0, L_0, t_0]$ is not a value state, then there exists a quantum closure $[Q_1, L_1, t_1]$ such that we have an evaluation step $[Q_0, L_0, t_0] \to_p [Q_1, L_1, t_1]$, but it is even said $[Q_1, L_1, t_1]$ is well-typed. This is legitimate since Selinger and Valiron show (strong) type preservation in simply typed *QLC* beforehand (Theorem 1.3.30 in [SV09]). In our above theorem, however, we can only claim $[Q_1, L_1, t_1]$ is a quantum closure since we do not establish a full type preservation result.

Secondly, Theorem 1.3.32 in [SV09] additionally establishes the following property. Let

$$\mathrm{res}_p\big([Q, L, t]\big) := \big\{[Q', L', t'] \bigm| \exists Q', L', t'.[Q, L, t] \to_p [Q', L', t']\big\}$$

be the set of all evaluation results that can be obtained from $[Q, L, t]$ in a single evaluation step with probability $p \in [0, 1]$. Then for each (well-typed) reducible quantum closure it holds

$$\sum_{p \in [0,1]} p \cdot \big|\mathrm{res}_p([Q, L, t])\big| = 1\ .$$

This is indeed an important property if one is interested in confluence and weak normalization of quantum closure reduction and properties connected with probabilistic reduction systems in general and in the particular context of quantum computation (where probability amplitudes occur rather than classical probabilities).

But since our focus is clearly on typing, and since our polymorphic approach is (in principle) not limited to the peculiarities of *QLC* and quantum computation in general, we do not go into the full formal details in the present work. Nevertheless, we claim that the described property holds for untyped *QLC* and for polymorphically typed *QLC* as well. The informal evidence is as follows. On the one hand, the call-by-value evaluation strategy we have established in *QLC*'s operational semantics guarantees that for each reducible quantum closure there is exactly one redex that can be reduced next. On the other hand, almost all probabilistic reduction rules are applied with probability 1. And for the only exception we have discussed after Definition 3.10 that there are two alternative evaluation steps whose respective probabilities sum up to 1.

4.6.2 Further towards type preservation

We have already met the type preservation property in several discussions throughout the present work. Type preservation is a very important property of type systems indeed. Again, we point out different grades at which type preservation could appear with respect to an evaluation $[Q, L, t] \to_p^* [Q', L', t']$:

weak form: $[Q, L, t] : \Phi$ and $[Q', L', t'] : \Psi$ are well-typed states but we know nothing about the relationship between type terms Φ and Ψ.

strong form: $[Q, L, t] : \Phi$ and $[Q', L', t'] : \Phi$ are well-typed states and can be assigned the same type.

intermediate form: $[Q, L, t] : \Phi$ and $[Q', L', t'] : \Phi'$ are well-typed states and we have $\Phi' \prec: \Phi$ or, a bit weaker, $\Phi' <: \Phi$.

For simply typed *QLC* Selinger and Valiron have confirmed the strong form of type preservation. And as already said before, we have pointed out some positive indications throughout our previous considerations on polymorphically typed *QLC* so that we may very well conjecture type preservation for our polymorphic extension. It is the purpose of this subsection to give a sketch of how we may approach the confirmation of this conjecture in future work. We do not work out the full details due to the limited scope of the present work and since the proof of the type preservation property still needs considerable work and a number of technical intermediate steps.

Let us start by answering the question why we insist on the distinction between strong and weak forms of type preservation in the context of typed *QLC*. Consider the evaluation step $[Q, L, (\lambda x{:}\Phi_x.x)\ v] \to_1 [Q, L, v]$ and derivation

$$\dfrac{\dfrac{\dfrac{\vdash x{:}\Phi_x \qquad \vdash \Phi_x \prec: \Psi_x}{x{:}\Phi_x \vdash x : \Psi_x}\ \text{(term variable)}}{\vdash (\lambda x{:}\Phi_x.x) : (\Phi_x \multimap \Psi_x)}\ \text{(linear term abstraction)} \qquad \vdash v : \Phi_x}{\vdash ((\lambda x{:}\Phi_x.x)\ v) : \Psi_x}\ \text{(term application)}\ .$$

On the one hand, derivation rule (linear term abstraction) requires that x' type Φ_x in the term context of its premise is taken over into term abstraction $(\lambda x{:}\Phi_x.x)$ and is also taken over as the left-hand side of function type $(\Phi_x \multimap \Psi_x)$ in the conclusion. Additionally, derivation rule (term application) requires that the type of v matches exactly the left-hand side of $(\Phi_x \multimap \Psi_x)$. On the other hand, rule (term variable) facilitates a certain amount of unparameterized polymorphism in its conclusion $x{:}\Phi_x \vdash x : \Psi_x$, and thus allows Φ_x and Ψ_x to differ. Hence, the intermediate form of type preservation applies in this particular case, since term application $((\lambda x{:}\Phi_x.x)\ v)$ of type Ψ_x reduces to value term v of type Φ_x with $\Phi_x \prec: \Psi_x$. However, when we aim at the strong form of type preservation, we need to get along with proved term $\vdash v : \Phi_x$ and well-typed state $[Q, L, v] : \Psi_x$ as strong type preservation implies. Obviously, we then need to derive a derivation rule

$$\dfrac{\Theta|\Gamma \vdash t : \Phi \qquad \Theta \vdash \Psi \qquad \Theta \vdash \Phi \prec: \Psi}{\Theta|\Gamma \vdash t : \Psi}\ \text{(generalize type)}\ .$$

(Selinger and Valiron in fact implicitly derive such a rule in Lemma 1.3.24(3) in [SV09].)

Such a derivation rule leads to interesting implications. Let $t = (\lambda y{:}\Phi_y.t')$ be a term abstraction. When we want to obtain proved term $\Theta|\Gamma \vdash t : (\Phi'_y \multimap \Psi')$ from proved term $\Theta|\Gamma \vdash t : (\Phi_y \multimap \Psi)$ and restricted proved subtype $\Theta \vdash (\Phi_y \multimap \Psi) \prec: (\Phi'_y \multimap \Psi')$ by application of (generalize type), then we are immediately confronted with the problem that we in general cannot derive $\Theta|\Gamma \vdash (\lambda y{:}\Phi_y.t') : (\Phi'_y \multimap \Psi')$, since type terms Φ_y and Φ'_y are not necessarily identical, but they have to be when we want to apply (linear term abstraction) or (nonlinear term abstraction). This means, the explicit type annotations in t are not appropriate any more. Hence, rule (generalize type) in the above form cannot work with explicit typing, and this consequently prevents us from establishing strong type preservation for explicitly typed *QLC*. One solution would be to modify (generalize type) in an inelegant way, where we replace function term t by a modified t'' and require that t'' is derived from t by appropriately replacing the respective type annotations (we take a look at this again a bit later). Clearly, the case is different if we switch over to implicit typing by completely omitting type annotations in function terms, since then rule (generalize type) should work without any problem.

Whichever approach we pursue, as soon as we try to derive rule (generalize type) we need to handle the following situation properly (presented in the implicitly typed approach):

$$\dfrac{\dfrac{\Theta|\Gamma, y{:}\Phi_y \vdash y : \Psi_y}{\Theta|\Gamma \vdash (\lambda y.y) : (\Phi_y \multimap \Psi_y)}\ \text{(linTmAbs)} \qquad \Theta \vdash (\Phi'_y \multimap \Psi'_y) \qquad \dfrac{\Theta \vdash \Phi'_y \prec: \Phi_y \qquad \Theta \vdash \Psi_y \prec: \Psi'_y}{\Theta \vdash (\Phi_y \multimap \Psi_y) \prec: (\Phi'_y \multimap \Psi'_y)}\ \text{(funST)}}{\Theta|\Gamma \vdash (\lambda y.y) : (\Phi'_y \multimap \Psi'_y)}\ \text{(generalize type)}\ .$$

Before we apply an inductive argument to premise $\Theta|\Gamma, y{:}\Phi_y \vdash y : \Psi_y$, we need to apply a yet to be derived rule

$$\frac{\Theta|\Gamma \vdash t : \Psi \qquad \dfrac{\Theta \vdash \Gamma_{sub}}{\Theta \vdash \Gamma_{sub} \prec: \Gamma}}{\Theta|\Gamma_{sub} \vdash t : \Psi}\ \text{(specialize term context)}\ ,$$

where $\Theta \vdash \Gamma_{sub} \prec: \Gamma$ means $|\Gamma_{sub}| = |\Gamma|$ and for each $x \in |\Gamma|$ the subtype-in-context $\Theta \vdash \Gamma_{sub}(x) \prec: \Gamma(x)$ is a restricted proved subtype. (Selinger and Valiron implicitly establish such a rule in Lemma 1.3.24(3) in [SV09] as well.)

Two last remarks are due regarding the proof of validity of the two derivation rules (generalize term) and (specialize term context). Firstly, both might be proved by induction on the derivation of proved terms $\Theta|\Gamma \vdash t : \Phi$ and $\Theta|\Gamma \vdash t : \Psi$, respectively. In the necessary base cases where these premises are derived by rules (new operator), (meas operator), (unitary operator) and (term variable) we need to apply Theorem 4.47. Secondly, the use of *restricted* proved subtypes is a crucial requirement, as we have already pointed out in the discussion of derivation rules (new operator), (meas operator), (unitary operator) following Definition 4.62. The issue becomes clear when we give up the restriction in the following example. Consider again the evaluation step $[Q, L, (\lambda x{:}\Phi_x.x)\ v] \to_1 [Q, L, v]$, but this time in connection with the following typing derivation (where we use implicit typing again):

$$\dfrac{\dfrac{\dfrac{\vdash x{:}(\Phi_1 \otimes \Phi_2) \qquad \dfrac{}{\vdash (\Phi_1 \otimes \Phi_2) <: \mathit{Top}}}{x{:}(\Phi_1 \otimes \Phi_2) \vdash x : \mathit{Top}}\ \text{(term variable)}}{\vdash (\lambda x.x) : ((\Phi_1 \otimes \Phi_2) \multimap \mathit{Top})}\ \text{(linear term abstraction)} \qquad \vdash v : (\Phi_1 \otimes \Phi_2)}{\vdash ((\lambda x.x)\ v) : \mathit{Top}}\ \text{(term application)}\ .$$

It is clear that v must be a pair term of the form $v = \langle v_1, v_2 \rangle$. However, for this pair term v we fail to derive type *Top*, since none of the derivation rules from Definitions 4.62 to 4.65 facilitates the derivation of type *Top* for a pair term. One way to solve this problem has already been proposed after Definition 4.62 and amounts to the introduction of an unrestricted variant of (generalize type) *by definition* (which uses general proved subtypes and not restricted ones) and not *by derivation*. This solution, however, might entail far reaching consequences that lie far beyond the scope of the present work. Another way to solve the described problem is the use of restricted subtypes, and we have taken this approach.

Let us now switch back to our explicitly typed approach towards polymorphically typed *QLC*. As we have just discussed, it could possibly help to turn our attention to the weak form of type preservation or to an intermediate form. However, there are also reasons speaking against this possibility. Consider the following typing derivation where we assume v and w to be value terms:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Theta \vdash x{:}(\Phi_1 \multimap \Phi_2) \qquad \dfrac{\Theta \vdash \Psi_1 \prec: \Phi_1 \qquad \Theta \vdash \Phi_2 \prec: \Psi_2}{\Theta \vdash (\Phi_1 \multimap \Phi_2) \prec: (\Psi_1 \multimap \Psi_2)}\ \text{(funST)}}{\Theta \vdash (\Psi_1 \multimap \Psi_2)}\ \text{(tmVar)}}{\Theta|x{:}(\Phi_1 \multimap \Phi_2) \vdash x : (\Psi_1 \multimap \Psi_2)} \qquad \Theta|\Gamma \vdash w : \Psi_1}\ \text{(tmApp)}}{\Theta|\Gamma, x{:}(\Phi_1 \multimap \Phi_2) \vdash (x\ w) : \Psi_2}}{\Theta|\Gamma \vdash \big(\lambda x{:}(\Phi_1 \multimap \Phi_2).(x\ w)\big) : ((\Phi_1 \multimap \Phi_2) \multimap \Psi_2)}\ \text{(linTmAbs)} \qquad \Theta|\Gamma' \vdash v : (\Phi_1 \multimap \Phi_2)}{\Theta|\Gamma, \Gamma' \vdash \big(\big(\lambda x{:}(\Phi_1 \multimap \Phi_2).(x\ w)\big)\ v\big) : \Psi_2}$$

Operational semantics then leads to an evaluation step

$$\Big[Q, L, \big(\lambda x{:}(\Phi_1 \multimap \Phi_2).(x\ w)\big)\ v\Big] \to_1 \Big[Q, L, v\ w\Big]\ .$$

For type preservation we now need to find a derivation for term-in-context $\Theta|\Gamma, \Gamma' \vdash (v\ w) : \Upsilon$ for some appropriate type term Υ. As a starting point we have proved terms $\Theta|\Gamma' \vdash v : (\Phi_1 \multimap \Phi_2)$ and $\Theta|\Gamma \vdash w : \Psi_1$ and restricted proved subtype $\Theta \vdash \Psi_1 \prec: \Phi_1$. But obviously, rule (term application) does not immediately facilitate a hypothetic derivation

$$\frac{\Theta|\Gamma' \vdash v : (\Phi_1 \multimap \Phi_2) \qquad \Theta|\Gamma \vdash w : \Psi_1}{\Theta|\Gamma, \Gamma' \vdash (v\ w) : \Upsilon}$$

since Φ_1 and Ψ_1 are not necessarily identical. Again, what would help is a derived rule (generalize type), but we have seen above that we then have to give up explicit typing. On the other hand, the introduction of an unrestricted rule (generalize type) of the form

$$\frac{\begin{array}{cc} & \Theta \vdash \Psi \\ \Theta|\Gamma \vdash t : \Phi & \Theta \vdash \Phi <: \Psi \end{array}}{\Theta|\Gamma \vdash t : \Psi}$$

as an additional derivation rule in our type system would solve this problem but with far reaching consequences, as we have already emphasized several times. Hence, also weakening of the formulation of type preservation does not help much (at least with the approaches discussed here).

A third way to a solution is offered by appropriate adaptation of typing annotations. To handle this, we would have to introduce a notion of *computational equivalence* that relates function terms s and t which differ only in their type annotations. For instance function terms $(\lambda x{:}\Phi_x.s')$ and $(\lambda x{:}\Psi_x.t')$ are computationally equivalent, if s' and t' are, even though Φ_x and Ψ_x may not be equal or not be subtypes of one another. Some authors (cf. [Pie02], pp. 109–110 and pp. 354–358, but also [SU06] and [Bar92]) define an erasure function in the spirit of the following exemplary cases:

$$\begin{aligned} \mathrm{erase}\big(x\big) &:= x \ , \\ \mathrm{erase}\big((\lambda x{:}\Phi_x.t)\big) &:= (\lambda x.\, \mathrm{erase}(t)) \ , \\ \mathrm{erase}\big((s\ t)\big) &:= (\mathrm{erase}(s)\ \mathrm{erase}(t)) \ , \\ \mathrm{erase}\big((\Lambda X{<:}\Phi_X.t)\big) &:= \mathrm{erase}(t) \ , \\ \mathrm{erase}\big((s\ \Phi)\big) &:= \mathrm{erase}(s) \ . \end{aligned}$$

Then computational equivalence of two function terms s and t amounts to the syntactical equality $\mathrm{erase}(s) \equiv \mathrm{erase}(t)$ modulo α-equivalence. Using this notion, we might be able to derive a modified (somewhat inelegant) variant of rule (generalize type) that is also valid for function terms with type annotations. However, this rule cannot be put in the form $\frac{\text{premises}}{\text{conclusion}}$ since it contains a nonconstructive existence statement:

> *Given proved term $\Theta|\Gamma \vdash t : \Phi$, proved type $\Theta \vdash \Phi'$ and proved subtype $\Theta \vdash \Phi \prec: \Phi'$, there exists a function term t' such that $\mathrm{erase}(t) \equiv_\alpha \mathrm{erase}(t')$ holds and proved term $\Theta|\Gamma \vdash t' : \Phi'$ is derivable.*

With this last point we leave the considerations on interaction of type preservation (in different grades) and the polymorphism induced by subtyping.

The thread of argument we tried to pick up until now would finally lead to a result that corresponds to a derived derivation rule

$$\frac{\Theta|!\Delta, \Gamma_1, z{:}\Phi_z \vdash t : \Psi \qquad \Theta|!\Delta, \Gamma_2 \vdash v : \Phi_z}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash t[v/z] : \Psi'} \text{ (value substitution)} \ ,$$

where v is a value term, and where we may or may not have some connection between types Ψ and Ψ', e.g. equality or a subtyping statement $\Psi' <: \Psi$.[53] The importance of such a result is evident since most of the basic reduction rules in Definition 4.74 are directly or indirectly based on substitution of free term variables. In order to handle the case of application of pair terms to pair abstractions we even need to extend rule (value substitution) to the setting of simultaneous substitutions of the sort

$$\big((\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t)\ \langle v, w\rangle\big) \to t[v/x, w/y] \ .$$

Another mechanism that we use in the basic reduction rules is substitution of free type variables. It appears in reduction rule

$$\big((\Lambda X{<:}\Phi_X.t)\ \Psi\big) \to t[\Psi/X]$$

dealing with type applications (cf. Definition 4.74). To handle the respective case in a proof of type preservation properly (using an inductive argument), we need to derive the following rule

[53] In Lemma 1.3.26 in [SV09] we can find a corresponding result for simply typed *QLC* fulfilling the case $\Psi = \Psi'$.

$$\frac{\Theta|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : !^n(\forall X{<:}\Phi_X.\Psi) \qquad \begin{matrix}\Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{matrix}}{\Theta|\Gamma \vdash t[\Upsilon/X] : !^n\Psi[\Upsilon/X]}\ \text{(type substitution)}\ .$$

This rule is very similar to derivation rule (type application) that we have introduced into our type system in Definition 4.63. The main difference is we actually perform substitution $t[\Upsilon/Z]$ in the conclusion, which corresponds to an adequate evaluation step (cf. Definition 4.74). To support the derivation of rule (type substitution), we have already done important preparatory steps in subsection 4.2.5 since we have derived rules (linear type substitution) and (nonlinear type substitution) there. In addition, we need a few more technical results, in order to establish (type substitution), but we do not explain these any further in the present work.

4.6.3 Linearity

We have already met the topic of linearity as a consequence of the no-cloning theorem (Theorem 2.5) in numerous places throughout section 4, and most of the important aspects have thus already been discussed exhaustively. Nevertheless, linearity has been a major point in motivating the use of typing to distinguish well-formed from ill-formed *QLC* function terms (also, but not exclusively, in a quantum physical sense). Thus, we undertake some conclusive considerations regarding linearity of well-typed function terms with respect to occurring term variables.

If we excluded case distinctions from our considerations, in other words, we would *not* use derivation rule (case distinction) to derive proved terms, then we could show for each in this way derived proved term $\Theta|\Gamma \vdash t : \Phi$ that the contained function term t is a *linear function term* in accordance to the following definition.

Definition 4.79 (linear function terms – strict version; adapted from Definition 10 in [FD01]).
A *linear function term* t is a typed *QLC* function term in which

- for each occurring term abstraction $(\lambda x{:}\Phi_x.t')$ in t, with linear type term Φ_x, term variable x does appear at most once as free term variable in the scope t' of the term abstraction;
- for each occurring pair abstraction $(\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t')$ in t, with linear type terms Φ_x (or Φ_y), term variables x (or y) does occur at most once in t';
- each free term variable in t has exactly one free occurrence.

Unfortunately, the case of (simply or polymorphically) typed *QLC* is more complicated, since we in general *allow* the use of case distinctions as a syntactical construct and thus also need the associated derivation rule (case distinction).

However, the claim we stated directly above Definition 4.79 is supported by the observation we make in the next proposition. This proposition also sheds some clarifying light on the notion of *linear function terms* in a weaker variant – weaker in the sense that it allows a certain amount of nonlinearity as it will become clear in the case concerning rule (case distinction) in the proof of Proposition 4.80 and in the discussion afterwards. (We do not give a formal definition of this weaker concept of linearity, since this definition would be too technical and we will not refer to it later.)

Proposition 4.80. *Let $\Theta|\Gamma \vdash t : \Phi$ be a proved term and $z \in \mathcal{V}_{term}$ a term variable with $z{:}\Phi_z \in \Gamma$ for a linear type term $\Phi_z \in \mathcal{T}_{type}$. Multiple free occurrences of z in t could only have been introduced by derivation rule (case distinction).*

Proof. In order to see this, we inspect all derivation rules for proved terms:

Rules (new operator), (meas operator), (unitary operator) and (*Unit* term): The conclusions of all these rules do not contain any term variables (neither free nor bound).

Rule (term variable): Assume $\Theta|\Gamma', x{:}\Phi_x \vdash x : \Psi_x$ has been derived using rule (term variable), where $x = z$ may hold or it may not. Obviously, term variable x appears exactly once in function term x (it does not matter whether Φ_x is linear or duplicable).

Rules (linear term abstraction), (nonlinear term abstraction), (linear pair abstraction) and (nonlinear pair abstraction): Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma, x{:}\Phi_x \vdash t' : \Psi'}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t') : (\Phi_x \multimap \Psi')} \text{ (linear term abstraction) ,}$$

where we assume $x \neq z$ without loss of generality (since we identify α-equivalent proved terms). This rule does not introduce any occurrences of free term variables. On the contrary, it turns all free occurrences of term variable x in function term t' into bound ones. The same is true for the other rules introducing abstractions binding term variables.

Rules (term application) and (pair term): Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash s' : !^n(\Phi' \multimap \Psi') \qquad \Theta|!\Delta, \Gamma_2 \vdash t' : \Phi'}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash (s'\ t') : \Psi'} \text{ (term application) .}$$

According to Definition 4.57 all type terms Φ_x with $x{:}\Phi_x \in\ !\Delta$ are nonlinear. Moreover, also by Definition 4.57, we assume $|\Gamma_1|$ and $|\Gamma_2|$ are disjoint whenever we write Γ_1, Γ_2 as in the conclusion of rule (term application). And since $|\Gamma_1|$ and $|\Gamma_2|$ are disjoint, we get $ftmv(s') \cap ftmv(t') = \emptyset$ by Lemma 4.67. Hence, each free term variable $z \in \mathcal{V}_{term}$ either occurs as often in function term $(s'\ t')$ as it does in function term s' and it does not occur in t' at all, or, symmetrically, z occurs as often in $(s'\ t')$ as it does in t' and then it does not appear in s' at all. An analogous argument holds for rule (pair term).

Rules (linear type abstraction), (nonlinear type abstraction) and (type application): Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma \vdash t' : !^n(\forall X{<:}\Phi_X.\Psi') \qquad \begin{array}{c}\Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array}}{\Theta|\Gamma \vdash (t'\ \Upsilon) : !^n\Psi'[\Upsilon/X]} \text{ (type application) .}$$

Obviously, this rule does not change function term t' and only adds a type term to obtain function term $(t'\ \Upsilon)$ embedded in a term-in-context. Hence, the occurrences of free term variables in t' are the same as in $(t'\ \Upsilon)$. Similar reasoning applies to rules (linear type abstraction) and (nonlinear type abstraction).

Rules (left injection) and (right injection): Suppose $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|\Gamma \vdash t' : !^n\Phi_l \qquad \Theta \vdash !^n\Phi_r}{\Theta|\Gamma \vdash inj_l(t') : !^n(\Phi_l \oplus \Phi_r)} \text{ (left injection) .}$$

Function term $inj_l(t')$ clearly has no new occurrences of free term variables in comparison to t' alone. A symmetrical argument applies to rule (right injection).

Rule (case distinction): Consider proved term $\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash (match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r))$ derived by

$$\frac{\begin{array}{cc}\multicolumn{2}{c}{\Theta|!\Delta, \Gamma_2 \vdash (\lambda x{:}\Phi_x.t_l) : !^m(!^n\Phi_x \multimap \Psi)} \\ \Theta|!\Delta, \Gamma_1 \vdash s : !^n(\Phi_x \oplus \Phi_y) & \Theta|!\Delta, \Gamma_2 \vdash (\lambda y{:}\Phi_y.t_r) : !^m(!^n\Phi_y \multimap \Psi)\end{array}}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash \big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big) : \Psi} \text{ (case distinction) ,}$$

where we assume $x \neq z \neq y$ without loss of generality. In analogy to rules (term application) and (pair term) we find

$$ftmv(s) \cap \big(ftmv(\lambda x{:}\Phi_x.t_l) \cup ftmv(\lambda y{:}\Phi_y.t_r)\big) = \emptyset\ .$$

Hence, a term variable z cannot occur freely in s and $(\lambda x{:}\Phi_x.t_l)$ at the same time, or in s and $(\lambda y{:}\Phi_y.t_r)$ at the same time. However, z may occur as free term variable in $(\lambda x{:}\Phi_x.t_l)$ and in $(\lambda y{:}\Phi_y.t_r)$, since both share term context Γ_2 in premises two and three of rule (case distinction).

Rule (recursion): Assume $\Theta|\Gamma \vdash t : \Phi$ has been derived by

$$\frac{\Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi) \vdash (\lambda x{:}\Phi_x.s) : !(\Phi_x \multimap \Psi) \qquad \Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi), \Gamma' \vdash t' : \Upsilon}{\Theta|!\Delta, \Gamma' \vdash (letrec\ f{:}!(\Phi_x \multimap \Psi) = (\lambda x{:}\Phi_x.s)\ in\ t') : \Upsilon}\ (\text{recursion})\ ,$$

where we assume $f \neq z$ without loss of generality. Then we see in the first premise that there are no free term variables of linear type allowed in $(\lambda x{:}\Phi_x.s)$, and thus the only occurrences of free term variables of linear type in $(letrec\ f{:}!(\Phi_x \multimap \Psi) = (\lambda x{:}\Phi_x.s)\ in\ t')$ may appear inside t'. Hence, in comparison to t' we get no new occurrences of free term variables in function term $(letrec\ f{:}!(\Phi_x \multimap \Psi) = (\lambda x{:}\Phi_x.s)\ in\ t')$.

□

Let us reconsider the case of rule (case distinction) in the above proof. Given a proved term $\Theta|\Gamma \vdash t : \Phi$, we know (by inspection of the derivation rules for proved terms) that for each subexpression t' of the form

$$t' = \big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big)$$

occurring in t, there must have been a proved term $\Theta'|\Gamma' \vdash t' : \Phi'$ that appeared during the derivation of $\Theta|\Gamma \vdash t : \Phi$. The above proof then states that a term variable z of linear type Φ_z cannot occur freely in s and $(\lambda x{:}\Phi_x.t_l)$ at the same time, or in s and $(\lambda y{:}\Phi_y.t_r)$ at the same time. However, z may occur as free term variable in $(\lambda x{:}\Phi_x.t_l)$ and in $(\lambda y{:}\Phi_y.t_r)$ at the same time.

Viewed from the perspective of operational semantics, this observation is of interest when we want to evaluate a quantum closure $\big[Q, |q_1, \ldots, q_k\rangle\,,\ match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big]$, where s shall either be of the form $s = inj_l(v)$ or of the form $s = inj_r(w)$ for value terms v and w. Then we have one of the two evaluation steps

$$\big[Q, |q_1, \ldots, q_k\rangle\,,\ match\ inj_l(v)\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big] \to_1 \big[Q, |q_1, \ldots, q_k\rangle\,, (\lambda x{:}\Phi_x.t_l)\ v\big]$$

or

$$\big[Q, |q_1, \ldots, q_k\rangle\,,\ match\ inj_r(w)\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big] \to_1 \big[Q, |q_1, \ldots, q_k\rangle\,, (\lambda y{:}\Phi_y.t_r)\ w\big]\ .$$

It does not matter whether one of the term variables $q_1, \ldots, q_k$ (of linear type $Qbit$), say q_j, does occur freely in $(\lambda x{:}\Phi_x.t_l)$ and $(\lambda y{:}\Phi_y.t_r)$ at the same time, since only one of the two function terms plays a role in the further course of evaluation. And thus only the occurrences of q_j in one of the two alternatives $((\lambda x{:}\Phi_x.t_l)\ v)$ and $((\lambda y{:}\Phi_y.t_r)\ w)$ contributes to the final result of the evaluation. On the other hand, it is now clear why q_j must not additionally appear in v or w as well. Note, however, that q_j might very well appear freely more than once in $(\lambda x{:}\Phi_x.t_l)$ or $(\lambda y{:}\Phi_y.t_r)$, but only as a result of nested case distinctions, as for example in proved term

$$\Gamma, q_j{:}Qbit \vdash \big(match\ inj_l(\langle\rangle)\ with\ \big(\lambda x{:}Unit.(match\ s\ with\ (\lambda x'{:}\Phi_{x'}.U\ q_j) \mid (\lambda y'{:}\Phi_{y'}.V\ q_j))\big) \mid \big(\ldots\big)\big) : Qbit\ ,$$

which is evaluated so that q_j contributes only once to the final result:
($U, V \in \mathcal{U}_1$ represent unary unitary operators)

$$\Big[Q, |\ldots, q_j, \ldots\rangle\,,\ match\ inj_l(\langle\rangle)\ with\ \big(\lambda x{:}Unit.(match\ s\ with\ (\lambda x'{:}\Phi_{x'}.U\ q_j) \mid (\lambda y'{:}\Phi_{y'}.V\ q_j))\big) \mid \big(\ldots\big)\Big]$$

$$\to_1^* \Big[Q, |\ldots, q_j, \ldots\rangle\,,\ match\ s\ with\ (\lambda x'{:}\Phi_{x'}.U\ q_j) \mid (\lambda y'{:}\Phi_{y'}.V\ q_j)\Big]$$

$$\to_p^* \begin{cases} \big[Q, |\ldots, q_j, \ldots\rangle\,,\ (\lambda x'{:}\Phi_{x'}.U\ q_j)\ v\big] & \text{if } s \to_p^* inj_l(v) \text{ with } v \in \mathcal{T}_{pValue} \text{ and } p \in [0,1] \\ \big[Q, |\ldots, q_j, \ldots\rangle\,,\ (\lambda y'{:}\Phi_{y'}.V\ q_j)\ w\big] & \text{if } s \to_p^* inj_r(w) \text{ with } w \in \mathcal{T}_{pValue} \text{ and } p \in [0,1] \end{cases}$$

$$\to_1^* \begin{cases} \big[Q_{Uq_j}, |\ldots, q_j, \ldots\rangle\,,\ q_j\big] \\ \big[Q_{Vq_j}, |\ldots, q_j, \ldots\rangle\,,\ q_j\big] \end{cases}$$

To contrast this *allowed* degree of nonlinearity, we give two simple example terms-in-context that are not derivable as proved terms due to a *forbidden* degree of nonlinearity:

- $f{:}(Qbit \multimap (Qbit \multimap Bit)), q{:}Qbit \vdash \big(\underline{(f\ (U\ q))}\ \underline{(V\ q)}\big) : Bit$,
- $\vdash \lambda q{:}Qbit.\big\langle \underline{(U\ q)}, \underline{(V\ q)} \big\rangle : (Qbit \multimap (Qbit \otimes Qbit))$.

In both cases, the underlined subexpressions cannot be combined into one proved term, since the responsible derivation rules (term application) and (pair term) prevent multiple occurrences of free term variables of linear type.

As a consequence of linearity our type system prevents implicit copying of data with linear type (with the "exception" of the alternative branches inside case distinctions). Consider for instance the untyped function term $(\lambda x.f\ x\ x)$ where f stands as a placeholder for a function term modeling a function. We have said in the very beginning of section 4 that the shape of this function term entails implicit copying in an evaluation step $[Q, L, (\lambda x.f\ x\ x)\ v] \rightarrow_1 [Q, L, f\ v\ v]$. Our type system can only assign a type to a function term $(\lambda x{:}\Phi_x.f\ x\ x)$ if type term Φ_x is duplicable, i.e. it needs to be of the form $\Phi_x = !\Phi'_x$. Then we end up with a proved term $\Theta|\Gamma \vdash (\lambda x{:}!\Phi'_x.f\ x\ x) : (!\Phi'_x \multimap \Psi)$. Hence, in order to assign a type to function term $((\lambda x.f\ x\ x)\ v)$, value term v must be of nonlinear type $!\Phi'_x$ as well. In this way our type system guarantees that only function terms of duplicable type can be subject to implicit copying.

Thus, well-typed *QLC* function terms are evaluated in accordance to the no-cloning theorem. Moreover, we have shown in subsection 4.2.3, and in particular in Theorem 4.39, that this property is not undermined by the mechanisms that we have established to introduce parametric polymorphism into *QLC*.

5 Conclusions and prospects

Let us briefly recapitulate what we have seen in the last few sections. After the introductory part we got acquainted with the basic formal notions of the scientific field of quantum computation in section 2. In the subsequent section we took a closer look at the *Quantum Lambda Calculus* – one particular formalism that models classical computations as well as quantum computations in an appropriately extended λ-calculus. The authors of *QLC* have equipped their approach to a quantum programming language with a type system that guarantees certain linearity constraints for well-typed *QLC* function terms. Moreover, well-typedness of a function term t implies that t is either a value term or t is reducible. And since typability is preserved by term reduction, this means simply typed *QLC* enjoys type safety. Building upon this background knowledge from research literature, we have then undertaken efforts to extend *QLC* with parametric (second-order) polymorphism in section 4. This extension and the detailed investigation of a few immediate consequences constitute the main focus of our work, and at the present point we just finished these investigation with considerations on type safety and linearity.

Throughout the text there have been a few direct and indirect hints towards further aspects (mostly of a technical nature) that might be worth further research. We point out some of these and even add a few more in the last part of this section. But before we come to the prospects part, we shall first contemplate what we have achieved up to now.

We have seen in the introductory section 1 that *QLC* is only one functional approach to quantum programming and that there is a variety of others. Most of the typed calculi among these languages are rooted in the λ-calculus and their type systems often incorporate the concepts of Girard's linear logic that embody a certain resource-awareness. Simply typed *QLC* in general and our polymorphic type system in particular are by no means an exception to this observation. However, there are of course characteristics that distinguish *QLC* from other calculi modeling classical computations as well as quantum computations – for instance the use of implicit linearity tracking at the level of function terms.[54] This stands in contrast to explicit linearity tracking that is often found in calculi with typing based on linear logic (cf. [Abr93] or [Bie94]). What we mean by tracking linearity in an explicit fashion is illustrated by syntactic constructs of the form

$$\text{copy } x \text{ as } y, z \text{ in } (f \text{ derelict}(y) \text{ derelict}(z)) \ ,$$

which corresponds to the *QLC* function term $(f\ x\ x)$.[55] In the explicit variant copying must be explicitly formulated by a *copy* construct, and the removal of nonlinearity from y's and z's types must be done explicitly by *derelict* constructs – one for each leading exponential – at the level of function terms. One consequence of this explicit approach is that nonlinear types $!\Phi$ and $!!\Phi$ are not conceived as being equivalent, which is in contrast to typed *QLC*. There are even calculi modeling quantum computations that are essentially untyped, but still explicitly track linearity in a way that reminds us of linear logic, e.g. the respective calculi investigated in [vTo04] and [DMZ11]. One could wonder whether polymorphic *QLC* in section 4 actually tracks linearity in an implicit manner, since we use explicit type annotations in which duplicability is marked by exponentials. However, there are no syntactical constructs outside these type annotations which facilitate linearity tracking. That means when we erased type annotations and thus switched to implicit typing, we would doubtless end up with implicit linearity tracking as well. Hence, the case is not entirely clear. More interesting to us than this conceptual controversy is the technique that needs to be employed to realize implicit linearity tracking. Technically speaking, we need to treat multiple leading exponentials in the same way as we treat a single one, i.e. type terms $!!\Phi$ and $!\Phi$ are equivalent when it comes down to duplicability of function terms that are assigned such types. In order to cope with this peculiarity, Selinger and Valiron have introduced subtyping. For our polymorphic extension we have successfully picked up this existing concept, combined it with universally quantified types to establish the already known concept of bounded quantification (see chapter 26 in [Pie02] for reference) and thus obtained the right tool to achieve one of our principal goals. By this we mean enforceable linearity (with respect to *Qbit* in particular, and *strictly linear type terms* in general) and the related result developed in subsection 4.2.3 which, together with the linearity result form subsection 4.6.3, confirms that even in the presence of parametric polymorphism quantum data is not cloned due to implicit copying when evaluating well-typed *QLC* states. However, as long as the type preservation property is only conjectured

[54] For further distinguishing characteristics with respect to related work, see section 1.3.8, pp. 156–159, in [SV09].

[55] The example is taken from [SV09], page 157, and uses the syntax of [Bie94]. For the related type derivation rules, see [Bie94], figure 3.1 on page 68, for complete reference, or [SV09], page 157, for a confined selection.

for our polymorphic extension of *QLC* and not formally proven, the previous sentence is only valid for single-step evaluations and not for evaluations in general. We have already started to bring a proof of type preservation on track in subsections 4.2.5 and 4.6.2, and have pointed out informal indications underpinning our conjecture in several places throughout the main text. But these efforts are of course by no means a substitute for a full formal proof. In addition to considerations on linearity of well-typed function terms, we have seen in subsection 4.6.1 that our type system does not assign types to error terms, these are terms that can neither be evaluated any further, nor do they represent reasonable evaluation results. In other words, we have confirmed polymorphically typed *QLC* enjoys the progress property.

But we have also made numerous technical achievements which emphasize some of the formal characteristics of polymorphic *QLC*. The most outstanding among these technical results is doubtless the theorem concerning transitivity of proved subtyping statements to which we have devoted subsection 4.2.4. On the one hand, the most demanding step towards final Theorem 4.47 has been taken in Lemma 4.46. On the other hand, the lemma itself and its proof give a deep insight into the structure of derivations of typing and subtyping judgements. And thus, for one part, we learn in the proof how we can distill the necessary typing environments from the term structure of type terms Φ and Ψ for which we want to derive a subtyping judgement $\Theta \vdash \Phi <: \Psi$. Of course, this only works when such a subtyping statement is derivable at all.

Let us now leave what we have done in the present work and take a look at what there is still to be done and what might promise interesting insights in future investigations.

At first and above all, the issue of type preservation should be settled. This is presumably not very hard to achieve but still requires some effort.

As a next step we might exploit the full benefits of subtyping by removing the posed restrictions on subtype expressions in some of the derivation rules.[56] We have already discussed right after Definition 4.62 that this entails the necessity of additional derivation rules to fully benefit from polymorphism induced by subtyping. This becomes even more interesting when we enrich *QLC* with more primitive datatypes. In fact Selinger and Valiron already have equipped simply typed *QLC* with lists in [SV09] and we have just omitted list constructs in the present work for simplification. Moreover, there are other worthwhile extensions, see chapter 11 in [Pie02] for inspirations. Among the possible candidates are certainly records. Let us stay with this suggestion for a moment and glance at possible consequences. Pierce distinguishes three "different sort[s] of flexibility in the use of records" ([Pie02], page 184) – *permutation subtyping*, *depth subtyping* and *width subtyping*. Each of these may be embodied by a specific derivation rule and two of these do not pose a major obstacle to our approach of typing. However, one of them, namely the *width subtyping* rule, indeed introduces a challenging aspect. It states that a record type $\{l_1{:}\Phi_1, \ldots, l_n{:}\Phi_n\}$ is a subtype of record type $\{l_1{:}\Phi_1, \ldots, l_n{:}\Phi_n, l_{n+1}{:}\Phi_{n+1}, \ldots, l_{n+m}{:}\Phi_{n+m}\}$, with $n, m \geq 1$ and where $l_1, \ldots, l_{n+m}$ denote the labels of the respective fields in the record data structure. This is so interesting, because in our work we rely on the fact that each subtype of $!\mathit{Top}$ is nonlinear and that $!\mathit{Top}$ is supertype of all nonlinear types. Now consider record type $\Psi := \{l_1{:}!\mathit{Bit}\}$, which we would certainly regard as nonlinear, and its relation to record type $\Upsilon := \{l_1{:}!\mathit{Bit}, l_2{:}\mathit{Qbit}\}$. Intuitively, we would have subtyping relationships $\Psi <: !\mathit{Top}$ and $\Upsilon \not<: !\mathit{Top}$. On the other hand, the above considerations on *width subtyping* tell us that $\Upsilon <: \Psi$ shall hold. But now transitivity of the subtype relation $<:$ says $\Upsilon <: \Psi \wedge \Psi <: !\mathit{Top} \Longrightarrow \Upsilon <: !\mathit{Top}$. But if $\{l_1{:}!\mathit{Bit}, l_2{:}\mathit{Qbit}\} <: !\mathit{Top}$ were a valid subtyping statement, then our mechanisms that make sure Qbit is never affected by exponentials would not work properly anymore. Hence, conflicts of this kind need to be resolved for a successful integration of record types into our framework.

In future investigations we should also consider other technical refinements. On the one hand, we have emphasized earlier that derivations of subtyping statements also admit types that we would actually rather prevent, such as $\Theta \vdash !\mathit{Qbit} <: !\mathit{Qbit}$. On the other hand, derivations of proved types are a bit too strict. Consider for instance the nonlinear function type $!(\mathit{Qbit} \multimap \mathit{Qbit})$. Our type derivation rules facilitate the derivation of a proved type $\vdash !(\mathit{Qbit} \multimap \mathit{Qbit})$. But we cannot derive $\vdash !(\forall X{<:}\mathit{Top}.(X \multimap X))$, since rule (linear-polymorphic type) does not allow the inside-out propagation of leading exponentials. This is not a problem for the mentioned example, since informally (in the spirit of operational semantics) we have the following evaluations of type applications

$!(\forall X{<:}\mathit{Top}.(X \multimap X))\ \mathit{Qbit} \rightsquigarrow !(\mathit{Qbit} \multimap \mathit{Qbit})$ for which $\vdash !(\forall X{<:}\mathit{Top}.(X \multimap X))$ is *not* derivable, and
$(\forall X{<:}\mathit{Top}.!(X \multimap X))\ \mathit{Qbit} \rightsquigarrow !(\mathit{Qbit} \multimap \mathit{Qbit})$ for which $\vdash (\forall X{<:}\mathit{Top}.!(X \multimap X))$ is derivable.

[56] Technically speaking, this means to allow proved subtypes instead of restricted ones in the derivation rules in Definition 4.62 and in rule (term variable).

However, problems arise in more complex situations, such as $\vdash !\big(Bit \otimes (\forall X{<:}Top.(X \multimap X))\big)$ which is not derivable in our system, but there is actually no serious reason why it should not be.

Of course, if we had done formal considerations on the completeness of our derivation rules, we would have discovered this inadequacy earlier and could have adjusted our typing rules accordingly. Hence, it is recommendable to make up for this in future work and scrutinize whether we actually can derive all types that are desirable and have excluded exactly those that necessarily need to be forbidden.

But there are more questions to be answered, and quite interesting ones, indeed:

- How expressive are untyped *QLC*, simply typed *QLC* and polymorphically typed *QLC* in comparison to one another and in comparison to other formalisms such as the classical λ-calculus (typed and untyped) and Deutsch's quantum Turing machine (cf. [Deu85] or [BV97])?

- What about decidability of type checking, typability and inhabitation?

- If typability is decidable, is it then possible to extend the type inference algorithm for simply typed *QLC* given by Selinger and Valiron (see section 1.4 in [SV09] for references) to the polymorphic setting?

Regarding decidability of type checking and typability, we need to consider the explicitly typed case and the implicitly typed one separately, of course. To get started with these questions, the exposition of algorithmic type inference of simply typed *QLC* in [Val04a] and [SV06] may be of help. For a general introduction to algorithmic type inference for the implicitly typed λ-calculus, see chapter 22 in [Pie02]. The first part of chapter 28 in the same textbook offers information on type inference in the presence of subtyping. As a third reference, the book chapter [Wal05] offers an introduction to algorithmic type checking for linear type systems. On the side of undecidability results, we find works by Wells, Schubert and Pierce who establish undecidability results for type checking and typability in system F in its implicitly typed ([Wel94]) and explicitly typed variants ([Sch98]) and in system F with bounded quantification ([Pie94]).[57] Let us also take a brief look at the inhabitation problem. To get started with a treatment of inhabitation, we find a proof of decidability (and *PSPACE* completeness) of inhabitation in the case of the simply typed λ-calculus and an undecidability result for inhabitation in system F (both explicitly typed) for instance in sections 4.2 and 11.6 in [SU06]. Regarding polymorphically typed *QLC*, there is a question closely connected to inhabitation: is each of the well-formed types inhabited? We can derive a proved term $\Theta|x{:}\Phi \vdash x : \Phi$ for each type term Φ for which we can derive proved type $\Theta \vdash \Phi$. The more interesting question is thus whether we can derive a *closed* proved term $\vdash t : \Phi$ for any proved type $\vdash \Phi$. A quite simple but presumably critical case is $\Phi = Top$ and, more generally, type terms containing *Top* as subexpression. These cases might get critical as long as we use restricted proved subtypes in rules (new operator), (meas operator), (unitary operator) and (term variable). If we abolished these restrictions, the case $\Phi = Top$ immediately turns into an uncritical one.

Finally we make the most vague suggestion for future considerations that aims at decoupling polymorphic typing with enforceable linearity from *QLC*, in order to make it generically applicable also to other calculi. Although we have widely isolated all investigations solely concerning type terms, proved types and proved subtypes from the investigations connected with function terms, we still needed function terms to introduce the concept of type application, for instance. On the other hand, not only *QLC* could benefit from a polymorphic extension, and our approach to parameterized linear polymorphism might be of help if appropriately adjusted. To support this sort of adaptation without the need for major modifications, we should adopt a different point of view towards syntactic constructs such as term constants *new* and *meas*, and also towards more complex syntactic constructs such as $inj_l(t)$ and $match\ s\ with\ (\lambda x.t_l) \mid (\lambda y.t_r)$. Instead of defining a "hard-wired" typing rule for each of them, one could use a general typing rule for function symbols, and reinterpret each of the mentioned syntactic constructs as application of a constant or function symbol and equip these symbols with type contexts and signatures similar to

$$\begin{aligned}
new &: [!(Bit \multimap Qbit){<:}X{<:}Top]\,;\, \to X\ , \\
meas &: [!(Qbit \multimap !Bit){<:}X{<:}Top]\,;\, \to X\ , \\
inj_l &: [X{<:}Top, Y{<:}Top]\,;\, !^n X \to !^n(X \oplus Y) && \text{for all } n \geq 0, \\
match &: [X{<:}Top, Y{<:}Top, Z{<:}Top]\,;\, !^n(X \oplus Y), !^m(!^n X \multimap Z), !^m(!^n Y \multimap Z) \to Z && \text{for all } m, n \geq 0.
\end{aligned}$$

[57] To gain a broader view on these results and related ones, sections 23.6 and 28.5 in [Pie02] are helpful.

General typing rules and signatures of constant and function symbols in this spirit are treated in chapters 4 to 6 in [Cro93] in the context of typed λ-calculi with simple types, second-order polymorphic types and higher-order polymorphic types. The mentioned textbook by Roy Crole might thus serve as a first inspiration in this direction.

Of course, a lot more enhancements, modifications, extensions and investigations on a meta level are conceivable. We leave this to the reader's sense of adventure.

A Qubit-by-qubit measurement of quantum registers

In this section we elaborate on the equivalence of qubit-by-qubit and all-at-once measurements of quantum registers. We therefor consider a quantum register of length n in state

$$|\varphi\rangle = \sum_{|b\rangle\in\mathfrak{B}_n} \alpha_b\,|b\rangle \qquad\qquad \text{with} \sum_{|b\rangle\in\mathfrak{B}_n} |\alpha_b|^2 = 1\ ,$$

where $\alpha_b \in \mathbb{C}$ for all $|b\rangle \in \mathfrak{B}_n$. Measuring the register qubit-by-qubit from leftmost to rightmost can be illustrated as a sequence

$$|\varphi\rangle \overset{\mathcal{M}_1}{\longrightarrow} |\varphi_1\rangle \overset{\mathcal{M}_2}{\longrightarrow} |\varphi_2\rangle \overset{\mathcal{M}_3}{\longrightarrow} \cdots \overset{\mathcal{M}_{n-1}}{\longrightarrow} |\varphi_{n-1}\rangle \overset{\mathcal{M}_n}{\longrightarrow} |\varphi_n\rangle\ ,$$

where $\mathcal{M}_j$ denotes a measurement of the j-th qubit with respect to the standard basis $\mathfrak{B}_1$.

The measurement results are obtained as follows:
(here $p_j(x_j)$ denotes the probability by which measuring the j-th qubit yields result $x_j \in \{0,1\}$, and we define sets $\mathcal{B}_k^n(x)$ as we have already done at the end of section 2, namely as sets of words of length n over $\{0,1\}$, where the k-th position is $x \in \{0,1\}$)

$$\begin{aligned}
|\varphi_1\rangle &= \frac{1}{\sqrt{p_1(x_1)}} \sum_{b\in\mathcal{B}_1^n(x_1)} \alpha_b\,|b\rangle & p_1(x_1) &= \sum_{b\in\mathcal{B}_1^n(x_1)} |\alpha_b|^2 \\
|\varphi_2\rangle &= \frac{1}{\sqrt{p_2(x_2)\cdot p_1(x_1)}} \sum_{\substack{b\in\mathcal{B}_1^n(x_1)\\ \cap\mathcal{B}_2^n(x_2)}} \alpha_b\,|b\rangle & p_2(x_2) &= \frac{1}{p_1(x_1)} \sum_{\substack{b\in\mathcal{B}_1^n(x_1)\\ \cap\mathcal{B}_2^n(x_2)}} |\alpha_b|^2 \\
&\ \ \vdots \\
|\varphi_n\rangle &= \frac{1}{\sqrt{\prod_{j=1}^n p_j(x_j)}} \underbrace{\sum_{b\in\bigcap_{j=1}^n \mathcal{B}_j^n(x_j)} \alpha_b\,|b\rangle}_{=\ \alpha_{x_1\ldots x_n}|x_1\ldots x_n\rangle} & p_n(x_n) &= \frac{1}{\prod_{j=1}^{n-1} p_j(x_j)} \underbrace{\sum_{b\in\bigcap_{j=1}^n \mathcal{B}_j^n(x_j)} |\alpha_b|^2}_{=\ |\alpha_{x_1\ldots x_n}|^2}\ . \qquad (2)
\end{aligned}$$

Following from this, a particular sequence of measurement results $x_1\ldots x_n \in \{0,1\}^{(n)}$ occurs with probability

$$\prod_{j=1}^n p_j(x_j) = \prod_{j=1}^{n-1} p_j(x_j) \cdot \underbrace{\frac{1}{\prod_{j=1}^{n-1} p_j(x_j)} |\alpha_{x_1\ldots x_n}|^2}_{\overset{(2)}{=}\ p_n(x_n)} = |\alpha_{x_1\ldots x_n}|^2\ , \qquad (3)$$

and leaves the register in state

$$|\varphi_n\rangle = \frac{1}{\sqrt{\prod_{j=1}^n p_j(x_j)}}\ \alpha_{x_1\ldots x_n}\,|x_1\ldots x_n\rangle \overset{(3)}{=} \frac{1}{\sqrt{|\alpha_{x_1\ldots x_n}|^2}}\ \alpha_{x_1\ldots x_n}\,|x_1\ldots x_n\rangle$$

which is (almost) what we would expect of an all-at-once measurement of the whole register with respect to the standard basis $\mathfrak{B}_n$. The complex scalar factor in front of $|x_1\ldots x_n\rangle$ is an "artifact" produced by our mathematical framework. In this case we may ignore it, since it holds

$$\left|\frac{\alpha_{x_1\ldots x_n}}{\sqrt{|\alpha_{x_1\ldots x_n}|^2}}\right|^2 = 1\ ,$$

which means $|\varphi_n\rangle$ is indistinguishable from $|x_1\ldots x_n\rangle$ by quantum mechanical measurement. And since this is the only way in which we can gain any knowledge about state $|\varphi_n\rangle$, we can very well accept the equivalence of $|\varphi_n\rangle$ and $|x_1\ldots x_n\rangle$.[58]

It is now clear that we can simulate all-at-once measurements of a whole quantum register (with respect to $\mathfrak{B}_n$) by step-by-step measurements of single qubits (with respect to $\mathfrak{B}_1$).

[58] Such an identification of states "up to a *global phase factor*" $\beta \in \mathbb{C}$ with $|\beta| = 1$ is a consequence of the standard framework of quantum computation and particularly of quantum mechanical measurement. For further details see for instance section 2.2.7 in [NC00], page 93.

B Application of arbitrary unitary operators in *QLC*

This section covers the details regarding application of unitary operators of arity $m \geq 2$ to an arbitrary set of m qubits in the operational semantics of *QLC*. In section 2 we have already seen unitary operators can be constructed by tensor product. Let $U : \mathcal{H}_{n'} \to \mathcal{H}_{n'}$ and $V : \mathcal{H}_{m'} \to \mathcal{H}_{m'}$ with $n', m' \geq 1$ be unitary operators. The tensor product then yields a unitary operator

$$U \otimes V : \mathcal{H}_{n'+m'} \to \mathcal{H}_{n'+m'};\ \ |\varphi\rangle \otimes |\psi\rangle \mapsto U\,|\varphi\rangle \otimes V\,|\psi\rangle\ \ .$$

Using this, we can handle the following simple cases of application of unitary operators in the operational semantics of *QLC*.

Consider for instance the following reduction of a quantum closure

$$\left[Q, |q_1, \ldots, q_n\rangle, U_m\ \langle q_{j_1}, \ldots, q_{j_m}\rangle\right] \to_1 \left[Q'', |q_1, \ldots, q_n\rangle, \langle q_{j_1}, \ldots, q_{j_m}\rangle\right]$$

with $2 \leq m \leq n$ and with pairwise distinct indices $j_1, \ldots, j_m$. According to the description of the corresponding reduction rule in Definition 3.10, Q'' is obtained from Q by application of U_m to qubits $j_1, \ldots, j_m$ inside quantum state Q, counted from left to right.

Suppose $q_{j_1}, \ldots, q_{j_m}$ link to neighboring qubits that appear in the appropriate order in Q, i.e. $j_1 = k, j_2 = k+1, \ldots, j_m = k+m-1$ for some k with $1 \leq k \leq n-(m-1)$. Then the application of U_m to qubits $j_1, \ldots, j_m$ can be performed by

$$Q'' = (Id_1^{\otimes k-1} \otimes U_m \otimes Id_1^{\otimes n-(k-1)-m})Q\ ,$$

analogously to the reduction rule for unary unitary operators.

But how can we handle cases where $q_{j_1}, \ldots, q_{j_m}$ do not link to neighboring qubits or to qubits that are not in the "right" order? This case clearly cannot be handled by tensor products alone. The solution to this problem is offered by unitary operators that implement adequate permutations of the components of basis vectors in standard bases $\mathfrak{B}_n$. That means, we need to construct two permutations σ, σ^{-1} of the first n positive natural numbers (see below for a formal definition) so that we get for each basis vector $|x_1 \ldots x_n\rangle \in \mathfrak{B}_n$

$$|x_1 \ldots x_n\rangle \overset{\sigma}{\longmapsto} |x_{j_1} \ldots x_{j_m} \ldots\rangle \overset{\sigma^{-1}}{\longmapsto} |x_1 \ldots x_n\rangle\ \ ,$$

i.e. $\sigma j_1 = 1, \ldots, \sigma j_m = m$ and $\sigma l \in \{m+1, \ldots, n\}$ for all $l \in \{1, \ldots, n\} \setminus \{j_1, \ldots, j_m\}$.

These permutations are used to construct unitary operators P_σ and $P_{\sigma^{-1}} = P_\sigma^{-1}$ for which

$$P_\sigma\,|x_1 \ldots x_n\rangle := |x_{\sigma 1} \ldots x_{\sigma n}\rangle \quad \text{and} \quad P_{\sigma^{-1}}\,|x_1 \ldots x_n\rangle := |x_{\sigma^{-1}1} \ldots x_{\sigma^{-1}n}\rangle$$

hold for all $|x_1 \ldots x_n\rangle \in \mathfrak{B}_n$ with $x_1, \ldots, x_n \in \{0, 1\}$.

Having these, we may realize the application of U_m to qubits $j_1, \ldots, j_m$ in Q by

$$Q'' := P_{\sigma-1}(U_m \otimes Id_1^{\otimes n-m})P_\sigma Q\ .$$

What remains to show is that operators P_σ and $P_{\sigma^{-1}}$ are unitary. We do this in two steps: at first we show arbitrary permutations π can be replaced by an appropriate sequence of composed transpositions $\tau_k \ldots \tau_1 = \pi$ for a certain $k \geq 1$, and in a second step we confirm a unitary operator P_τ realizing a transposition of basis vector components is unitary. Consequently, we may conclude $P_\pi := P_{\tau_k} \ldots P_{\tau_1}$ is unitary for any permutation π, since we know from section 2 that composition of unitary operators yields again a unitary operator.

Definition B.1 (permutations, transpositions).

A *permutation on the first n positive natural numbers* is a bijective mapping $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}; l \mapsto \pi l$.

We call a permutation τ on the first n positive natural numbers *transposition* if it exhibits one of the following two properties:

- either τ is the identity mapping,
- or $\tau k \neq k$ holds for exactly two distinct $k \in \{1, \ldots, n\}$.

Proposition B.2. *Let $(x_1, \ldots, x_n) \in \{0,1\}^n$ be an n-tuple of zeros and ones ($n \geq 2$), and let π be a permutation of the first n positive natural numbers. We can find a sequence $\tau_1, \ldots, \tau_{n-1}$ of transpositions on the first n positive natural numbers, such that*

$$(x_{\tau_{n-1}\ldots\tau_1 1}, \ldots, x_{\tau_{n-1}\ldots\tau_1 n}) = (x_{\pi 1}, \ldots, x_{\pi n}) \ .$$

Proof. We construct transpositions $\tau_1, \ldots, \tau_{n-1}$ as follows (for all l with $1 \leq l \leq n$):

$$\tau_j l := \begin{cases} \pi j & \text{if } l = \tau_{j-1} \ldots \tau_1 j, \\ \tau_{j-1} \ldots \tau_1 j & \text{if } l = \pi j, \\ l & \text{otherwise.} \end{cases}$$

We show by induction on $m \geq 0$ that $\tau_m \ldots \tau_1 l = \pi l$ holds for each l with $1 \leq l \leq m$ and for all $m \leq n-1$.

Base case: Let $m = 1$. Then we have $\tau_1 1 = \pi 1$ by definition of τ_1.

Induction case: Consider m with $1 < m \leq n-1$. In this case we get

$$\tau_m l = \begin{cases} \pi m & \text{if } l = \tau_{m-1} \ldots \tau_1 m, \\ \tau_{m-1} \ldots \tau_1 m & \text{if } l = \pi m, \\ l & \text{otherwise.} \end{cases}$$

(Please note, the two upper cases in the above equation coincide in case of $\pi m = \tau_{m-1} \ldots \tau_1 m$. Then the constructed transposition is the identity mapping.)

By induction, we get

$$\begin{gathered} \tau_{m-1} \ldots \tau_1 1 = \pi 1 \\ \tau_{m-1} \ldots \tau_1 2 = \pi 2 \\ \vdots \\ \tau_{m-1} \ldots \tau_1 (m-1) = \pi (m-1) \end{gathered}$$

where all $\pi 1, \ldots, \pi(m-1)$ are pairwise distinct, different from πm (due to surjectivity of π) and also unequal to $\tau_{m-1} \ldots \tau_1 m$ (due to surjectivity of $\tau_{m-1}, \ldots, \tau_1$). Thus, these values remain unchanged by τ_m, and it hence holds $\tau_m \ldots \tau_1 l = \pi l$ for all l with $1 \leq l \leq m-1$. Moreover, the definition of τ_m immediately yields $\tau_m \tau_{m-1} \ldots \tau_1 m = \pi m$. Putting these facts together, we obtain $\tau_m \ldots \tau_1 l = \pi l$ for all l, m with $1 \leq l \leq m \leq n-1$.

$\Diamond$

Finally, injectivity of π and of all the $\tau_m, \ldots, \tau_1$ entails $\pi n = \tau_{n-1} \ldots \tau_1 n$.
Hence, we get the equality $(x_{\tau_{n-1}\ldots\tau_1 1}, \ldots, x_{\tau_{n-1}\ldots\tau_1 n}) = (x_{\pi 1}, \ldots, x_{\pi n})$. $\square$

Proposition B.3. *If $\tau : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a transposition, then the linear operator $P_\tau : \mathcal{H}_n \to \mathcal{H}_n; |x_1 \ldots x_n\rangle \mapsto |x_{\tau 1} \ldots x_{\tau n}\rangle$ is unitary.*

Proof. Let $\tau : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a transposition and let $P_\tau : \mathcal{H}_n \to \mathcal{H}_n$ be a linear operator with $P_\tau |x_1 \ldots x_n\rangle := |x_{\tau 1} \ldots x_{\tau n}\rangle$. If τ is the identity mapping, then $P_\tau = Id_n$ is the identity operator on Hilbert space $\mathcal{H}_n$. And this is clearly unitary since it preserves length. If τ is not the identity mapping, then (by definition of transpositions) there exist exactly two distinct $k, k' \in \{1, \ldots, n\}$ such that $k \neq \tau k$ and $k' \neq \tau k'$. We assume $k < k'$ (without loss of generality). Consequently, we know $\tau k = k'$ and $\tau k' = k$, and it moreover immediately follows that τ is self-inverse. Hence, when we apply P_τ to a basis vector $|x_1 \ldots x_n\rangle \in \mathfrak{B}_n$ from the 2^n-dimensional standard basis, we end up with

$$P_\tau |x_1 \ldots x_k \ldots x_{k'} \ldots x_n\rangle = |x_1 \ldots x_{k'} \ldots x_k \ldots x_n\rangle$$

and

$$P_\tau |x_1 \ldots x_{k'} \ldots x_k \ldots x_n\rangle = |x_1 \ldots x_k \ldots x_{k'} \ldots x_n\rangle \ .$$

This entails that the set $\{P_\tau \left|b\right\rangle \mid \left|b\right\rangle \in \mathfrak{B}_n\}$ is equal to $\mathfrak{B}_n$. In other words, P_τ is bijective and it at least preserves the length of basis vectors from $\mathfrak{B}_n$. Keeping this in mind, we now take a look at the quadratic length of state vector $P_\tau \left|\varphi\right\rangle$ for an arbitrary state $\left|\varphi\right\rangle \in \mathcal{H}_n$:

$$\begin{aligned}
\| P_\tau \left|\varphi\right\rangle \|^2 &= \left\| P_\tau \sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_b \left|b\right\rangle \right\|^2 \\
&= \left\langle P_\tau \sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_b \left|b\right\rangle \,\middle|\, P_\tau \sum_{\left|b'\right\rangle \in \mathfrak{B}_n} \alpha_{b'} \left|b'\right\rangle \right\rangle \\
&= \sum_{\left|b'\right\rangle \in \mathfrak{B}_n} \alpha_{b'} \left\langle \sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_b P_\tau \left|b\right\rangle \,\middle|\, P_\tau \left|b'\right\rangle \right\rangle \\
&= \sum_{\left|b'\right\rangle \in \mathfrak{B}_n} \alpha_{b'} \left(\sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_b \left\langle P_\tau \left|b'\right\rangle \,\middle|\, P_\tau \left|b\right\rangle \right\rangle \right)^* \\
&= \sum_{\left|b'\right\rangle \in \mathfrak{B}_n} \sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_{b'} \alpha_b^* \left\langle P_\tau \left|b\right\rangle \,\middle|\, P_\tau \left|b'\right\rangle \right\rangle .
\end{aligned}$$

Due to surjectivity of P_τ and orthonormality of $\mathfrak{B}_n$, we have

$$\left\langle P_\tau \left|b\right\rangle \middle| P_\tau \left|b'\right\rangle \right\rangle = \begin{Bmatrix} 0 & \text{if } \left|b\right\rangle \neq \left|b'\right\rangle \\ 1 & \text{otherwise} \end{Bmatrix} = \left\langle b \middle| b' \right\rangle ,$$

and hence, we furthermore get

$$\begin{aligned}
\| P_\tau \left|\varphi\right\rangle \|^2 &= \sum_{\left|b'\right\rangle \in \mathfrak{B}_n} \sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_{b'} \alpha_b^* \left\langle P_\tau \left|b\right\rangle \,\middle|\, P_\tau \left|b'\right\rangle \right\rangle \\
&= \sum_{\left|b'\right\rangle \in \mathfrak{B}_n} \sum_{\left|b\right\rangle \in \mathfrak{B}_n} \alpha_{b'} \alpha_b^* \left\langle b \,\middle|\, b' \right\rangle \\
&= \| \left|\varphi\right\rangle \|^2 .
\end{aligned}$$

That means P_τ preserves length and is thus a unitary operator. □

C Supplementary material for section 4.5

Here we collect a number of type and term derivations and evaluations of function terms that have been omitted in section 4.5 in the main text to save space.

C.1 Details concerning Example 4.75

In the following type derivations, we use the following abbreviations:

$$
\begin{aligned}
\mathrm{coin}_{poly} &:\equiv \big(\lambda z{:}\mathit{Unit}.\mathrm{meas}\ (H\ (\mathrm{new}\ 0))\big)\\
\Phi_f &:= (\forall Y{<:}!\mathit{Top}.(!Y \multimap !!(!X \otimes Y)))\\
X^{!\otimes n} &:= !(!X \otimes !(\ldots !(!X \otimes !X)\ldots))\\
(!\mathit{Bit})^{!\otimes n} &:= !(!!\mathit{Bit} \otimes !(\ldots !(!!\mathit{Bit} \otimes !!\mathit{Bit})\ldots))\\
\ulcorner n-1 \urcorner^{!\otimes} &:\equiv \big(\Lambda X{<:}!\mathit{Top}.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}(\ldots(f\ !X\ x)\ldots))\big)
\end{aligned}
$$

C.1.1 Type derivations

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}}\,(\mathit{Top}\text{ type})}{X{<:}!\mathit{Top} \vdash !\mathit{Top}}}{X{<:}!\mathit{Top} \vdash !\mathit{Top} <: !\mathit{Top}}\,(<:\text{ refl}) \qquad \dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}, Y{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top}, Y{<:}!\mathit{Top} \vdash !Y}\,(\text{nlTyVar}) \qquad \dfrac{\dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}, Y{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top}, Y{<:}!\mathit{Top} \vdash !!!X}\,(\text{nlTyVar}) \qquad \dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}, Y{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top}, Y{<:}!\mathit{Top} \vdash !!Y}\,(\text{nlTyVar})}{X{<:}!\mathit{Top}, Y{<:}!\mathit{Top} \vdash !!(!X \otimes Y)}\,(\text{prodTy})}{X{<:}!\mathit{Top}, Y{<:}!\mathit{Top} \vdash !(!Y \multimap !!(!X \otimes Y))}\,(\text{funTy})}{X{<:}!\mathit{Top} \vdash !\underbrace{(\forall Y{<:}!\mathit{Top}.(!Y \multimap !!(!X \otimes Y)))}_{=\,\Phi_f}}\,(\text{nlPolyTy})\ \boxed{1}
$$

$$
\dfrac{\dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !!X}\,(\text{nlTyVar}) \qquad \dfrac{\dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !!!X}\,(\text{nlTyVar}) \qquad \dfrac{\dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !^{n-1}X}\,(\text{nlTyVar}) \qquad \dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !^{n}X}\,(\text{nlTyVar}) \qquad \dfrac{\dfrac{}{\vdash X{<:}!\mathit{Top}}\,(\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !^{n}X}\,(\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !^{n-1}(!X \otimes !X)}\,(\text{prodTy})}{X{<:}!\mathit{Top} \vdash !^{n-2}(!X \otimes !(!X \otimes !X))}\,(\text{prodTy}) \qquad \ddots}{X{<:}!\mathit{Top} \vdash !(!X \otimes X^{!\otimes n-3})}\,(\text{prodTy})}{X{<:}!\mathit{Top} \vdash !\underbrace{(!X \otimes X^{!\otimes n-2})}_{=\,X^{!\otimes n-1}}}\ \boxed{2}
$$

$$\dfrac{\dfrac{\boxed{1}\ X{<:}!\mathit{Top} \vdash f{:}!\Phi_f \qquad \boxed{1}\ X{<:}!\mathit{Top} \vdash !\Phi_f \qquad \dfrac{\boxed{1}\ X{<:}!\mathit{Top} \vdash !\Phi_f}{X{<:}!\mathit{Top} \vdash !\Phi_f \prec: !\Phi_f}\ (<:\ \text{refl})}{X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash f : !(\forall Y{<:}!\mathit{Top}.(!Y \multimap !!(!X \otimes Y)))}\ (\text{tmVar}) \qquad \boxed{2}\ X{<:}!\mathit{Top} \vdash X^{!\otimes n-1} \qquad \dfrac{\dfrac{\boxed{2}\ X{<:}!\mathit{Top} \vdash X^{!\otimes n-1}}{X{<:}!\mathit{Top} \vdash X^{!\otimes n-1} <: \mathit{Top}}\ (\mathit{Top}\ \text{supTy})}{X{<:}!\mathit{Top} \vdash X^{!\otimes n-1} <: !\mathit{Top}}\ (!\ \text{right})}{X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash (f\ X^{!\otimes n-1}) : !(!X^{!\otimes n-1} \multimap !\underbrace{!(!X \otimes X^{!\otimes n-1})}_{= X^{!\otimes n}})}\ (\text{tyApp})\ \boxed{3}$$

$$\dfrac{\dfrac{\dfrac{\boxed{1}\ X{<:}!\mathit{Top} \vdash f{:}!\Phi_f \qquad \dfrac{\boxed{1}\ X{<:}!\mathit{Top} \vdash !\Phi_f}{X{<:}!\mathit{Top} \vdash !\Phi_f \prec: !\Phi_f} \qquad \dfrac{\vdash X{<:}!\mathit{Top}}{X{<:}!\mathit{Top} \vdash !X}\ (\text{nlTyVar})}{X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash f : !(\forall Y{<:}!\mathit{Top}.(!Y \multimap !!(!X \otimes Y)))} \qquad \dfrac{\vdash X{<:}!\mathit{Top}}{X{<:}!\mathit{Top} \vdash !X} \qquad \dfrac{\dfrac{\vdash X{<:}!\mathit{Top}}{X{<:}!\mathit{Top} \vdash !X}\ (\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !X <: \mathit{Top}}\ (!\ \text{right})}{X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash (f\ !X) : !(!!X \multimap !!(!X \otimes !X))}\ (\text{tyApp}) \qquad \dfrac{X{<:}!\mathit{Top} \vdash x{:}!X \qquad \dfrac{\vdash X{<:}!\mathit{Top}}{X{<:}!\mathit{Top} \vdash !!X} \qquad \dfrac{\dfrac{\vdash X{<:}!\mathit{Top}}{X{<:}!\mathit{Top} \vdash !X}\ (\text{nlTyVar})}{X{<:}!\mathit{Top} \vdash !X \prec: !!X}\ (!\ \text{right})}{X{<:}!\mathit{Top}|x{:}!X \vdash x : !!X}\ (\text{tmVar})}{X{<:}!\mathit{Top}|f{:}!\Phi_f, x{:}!X \vdash (f\ !X\ x) : !!(!X \otimes !X)}\ (\text{tmApp})\ \boxed{4}$$

$$\dfrac{\dfrac{X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash (f\ X^{!\otimes 2}) : !(!X^{!\otimes 2} \multimap !X^{!\otimes 3}) \qquad \boxed{4}\ X{<:}!\mathit{Top}|f{:}!\Phi_f, x{:}!X \vdash (f\ !X\ x) : !X^{!\otimes 2}}{X{<:}!\mathit{Top}|f{:}!\Phi_f, x{:}!X \vdash (f\ X^{!\otimes 2}\ (f\ !X\ x)) : !X^{!\otimes 3}}\ (\text{tmApp})}{\boxed{3}\ X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash (f\ X^{!\otimes n-1}) : !(!X^{!\otimes n-1} \multimap !X^{!\otimes n}) \qquad X{<:}!\mathit{Top}|f{:}!\Phi_f, x{:}!X \vdash (f\ X^{!\otimes n-2}\ (\ldots(f\ !X\ x)\ldots)) : !X^{!\otimes n-1}}\ (\text{tmApp})\ \ddots}{X{<:}!\mathit{Top}|f{:}!\Phi_f, x{:}!X \vdash (f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\ldots(f\ !X\ x)\ldots))) : !X^{!\otimes n}}\ (\text{tmApp})\ \boxed{5}$$

$$\dfrac{\dfrac{\vdash \emptyset}{\vdash !\mathit{Top}}}{\vdash !\mathit{Top} <: !\mathit{Top}} \qquad \dfrac{\dfrac{\boxed{5}\ X{<:}!\mathit{Top}|f{:}!\Phi_f, x{:}!X \vdash (f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\ldots(f\ !X\ x)\ldots))) : !X^{!\otimes n}}{X{<:}!\mathit{Top}|f{:}!\Phi_f \vdash (\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\ldots(f\ !X\ x)\ldots))) : (!X \multimap !X^{!\otimes n})}\ (\text{linTmAbs}) \qquad \mathit{ftmv}(\ldots) \cap |\emptyset| = \emptyset}{X{<:}!\mathit{Top}|\emptyset \vdash (\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\ldots(f\ !X\ x)\ldots))) : !(!\Phi_f \multimap (!X \multimap !X^{!\otimes n}))}\ (\text{nlTmAbs}) \qquad X \notin \mathit{ftyv}(\emptyset)}{\vdash \underbrace{(\Lambda X{<:}!\mathit{Top}.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\ldots(f\ !X\ x)\ldots)))}_{\equiv\ \ulcorner n-1 \urcorner^{!\otimes}} : (\forall X{<:}!\mathit{Top}.!(!\underbrace{(\forall Y{<:}!\mathit{Top}.(!Y \multimap !!(!X \otimes Y)))}_{= \Phi_f} \multimap (!X \multimap !X^{!\otimes n})))}\ (\text{nlTyAbs})\ \boxed{6}$$

$$\dfrac{\dfrac{\vdash z{:}\mathit{Unit} \qquad \prec\!: (\mathit{Qbit} \multimap !!\mathit{Bit})}{z{:}\mathit{Unit} \vdash \mathrm{meas} : (\mathit{Qbit} \multimap !!\mathit{Bit})}\ (\mathrm{measOp}) \qquad \dfrac{\vdash \emptyset \qquad \dfrac{\vdash !(\mathit{Qbit} \multimap \mathit{Qbit})}{\prec\!: (\mathit{Qbit} \multimap \mathit{Qbit})} \qquad H \in \mathcal{U}_1}{\vdash H : (\mathit{Qbit} \multimap \mathit{Qbit})}\ (\mathrm{unitaryOp}) \qquad \dfrac{\vdash \emptyset \qquad \dfrac{\vdash !(\mathit{Bit} \multimap \mathit{Qbit})}{\prec\!: (\mathit{Bit} \multimap \mathit{Qbit})}\ (\mathrm{newOp})}{\vdash \mathrm{new} : (\mathit{Bit} \multimap \mathit{Qbit})} \qquad \vdash 0 : \mathit{Bit}}{\vdash (\mathrm{new}\ 0) : \mathit{Qbit}}\ (\mathrm{tmApp})}{\vdash (H\ (\mathrm{new}\ 0)) : \mathit{Qbit}}\ (\mathrm{tmApp})}{z{:}\mathit{Unit} \vdash (\mathrm{meas}\ (H\ (\mathrm{new}\ 0))) : !!\mathit{Bit}}\ \boxed{7}$$

$$\dfrac{\boxed{7}\ z{:}\mathit{Unit} \vdash (\mathrm{meas}\ (H\ (\mathrm{new}\ 0))) : !!\mathit{Bit} \qquad \mathit{ftmv}\big((\mathrm{meas}\ (H\ (\mathrm{new}\ 0)))\big) \cap |z{:}\mathit{Unit}| = \emptyset}{\vdash \underbrace{\big(\lambda z{:}\mathit{Unit}.(\mathrm{meas}\ (H\ (\mathrm{new}\ 0)))\big)}_{\equiv\ \mathrm{coin}_{\mathrm{poly}}} : !(\mathit{Unit} \multimap !!\mathit{Bit})}\ (\mathrm{nlTmAbs})\ \boxed{8}$$

$$\dfrac{\boxed{8}\ \vdash \mathrm{coin}_{\mathrm{poly}} : !(\mathit{Unit} \multimap !!\mathit{Bit}) \qquad \dfrac{\vdash \emptyset}{\vdash \langle\rangle : \mathit{Unit}}\ (\mathit{Unit}\ \mathrm{term})}{\vdash (\mathrm{coin}_{\mathrm{poly}}\ \langle\rangle) : !!\mathit{Bit}}\ (\mathrm{tmApp})\ \boxed{9}$$

$$\dfrac{Y{<\!:}!\mathit{Top}|\emptyset \vdash \mathrm{coin}_{\mathrm{poly}} : !(\mathit{Unit} \multimap !!!!\mathit{Bit}) \qquad \dfrac{Y{<\!:}!\mathit{Top} \vdash \emptyset}{Y{<\!:}!\mathit{Top}|\emptyset \vdash \langle\rangle : \mathit{Unit}}\ (\mathit{Unit}\ \mathrm{term})}{Y{<\!:}!\mathit{Top}|\emptyset \vdash (\mathrm{coin}_{\mathrm{poly}}\ \langle\rangle) : !!!!\mathit{Bit}}\ (\mathrm{tmApp})\ \boxed{10}$$

$$\dfrac{\dfrac{\dfrac{\vdash \emptyset}{\vdash !Top}}{\vdash !Top <: !Top} \qquad \dfrac{\boxed{10}\ Y{<:}!Top|\emptyset \vdash (\mathrm{coin}_{\mathrm{poly}}\ \langle\rangle) : !!!!Bit \qquad \dfrac{Y{<:}!Top \vdash y{:}!Y \qquad \dfrac{\vdash Y{<:}!Top}{Y{<:}!Top \vdash !!Y} \qquad \dfrac{\dfrac{\vdash Y{<:}!Top}{Y{<:}!Top \vdash !Y}}{Y{<:}!Top \vdash !Y \prec: !!Y}}{Y{<:}!Top|y{:}!Y \vdash y : !!Y}\ (\text{pairTm})}{Y{<:}!Top|y{:}!Y \vdash \langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle : !!(!!Bit \otimes Y)} \qquad ftmv(\dots) \cap |\emptyset| = \emptyset}{Y{<:}!Top|\emptyset \vdash (\lambda y{:}!Y.\langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle) : !(!Y \multimap !!(!!Bit \otimes Y))}\ (\text{nlTmAbs}) \qquad Y \notin ftyv(\emptyset)}{\vdash (\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle) : !(\forall Y{<:}!Top.(!Y \multimap !!(!!Bit \otimes Y)))\ \boxed{11}}\ (\text{nlTyAbs})$$

$$\dfrac{\boxed{6} \vdash \big(\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots))\big) : (\forall X{<:}!Top.!(!\overbrace{(\forall Y{<:}!Top.(!Y \multimap !!(!X \otimes Y)))}^{=\Phi_f} \multimap (!X \multimap !X^{!\otimes n}))) \qquad \vdash !Bit <: !Top}{\vdash \big(\underbrace{(\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots)))}_{\equiv\ \ulcorner n-1 \urcorner^{!\otimes}}\ !Bit\big) : !(!(\forall Y{<:}!Top.(!Y \multimap !!(!!Bit \otimes Y))) \multimap (!!Bit \multimap !(!Bit)^{!\otimes n}))\ \boxed{12}}\ (\text{tyApp})$$

$$\dfrac{\boxed{12} \vdash \big((\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots)))\ !Bit\big) : !(!(\forall Y{<:}!Top.(!Y \multimap !!(!!Bit \otimes Y))) \multimap (!!Bit \multimap !(!Bit)^{!\otimes n})) \qquad \boxed{11} \vdash (\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle) : !(\forall Y{<:}!Top.(!Y \multimap !!(!!Bit \otimes Y)))}{\vdash \Big(\underbrace{(\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots)))}_{\equiv\ \ulcorner n-1 \urcorner^{!\otimes}}\ !Bit\ \big(\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle\big)\Big) : (!!Bit \multimap !(!Bit)^{!\otimes n})\ \boxed{13}}\ (\text{tmApp})$$

$$\dfrac{\boxed{13} \vdash \Big(\big(\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots))\big)\ !Bit\ \big(\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle\big)\Big) : (!!Bit \multimap !(!Bit)^{!\otimes n}) \qquad \boxed{9} \vdash (\mathrm{coin}_{\mathrm{poly}}\ \langle\rangle) : !!Bit}{\vdash \underbrace{\Big(\underbrace{(\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots)))}_{\equiv\ \ulcorner n-1 \urcorner^{!\otimes}}\ !Bit\ \big(\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \mathrm{coin}_{\mathrm{poly}}\ \langle\rangle,\ y\rangle\big)\ \big(\mathrm{coin}_{\mathrm{poly}}\ \langle\rangle\big)\Big)}_{\equiv\ t_{\mathrm{poly}}} : !(!Bit)^{!\otimes n}}\ (\text{tmApp})$$

C.1.2 Function term evaluations

Untyped case:

The following informal full evaluation of untyped *QLC* function term t_{toss} (cf. Example 4.75) may be interpreted as n tosses of a fair (quantum) coin.[59]

$$
\begin{aligned}
t_{\text{toss}} &\equiv \overbrace{\big(\lambda f.\lambda x.f\ (\dots(f\ x)\dots)\big)}^{\equiv\ulcorner n-1\urcorner}\ \big(\lambda y.\langle \text{coin}\ \langle\rangle,\ y\rangle\big)\ \big(\text{coin}\ \langle\rangle\big) \\
&\longrightarrow^{*} \big(\lambda f.\lambda x.f\ (\dots(f\ x)\dots)\big)\ \big(\lambda y.\langle \text{coin}\ \langle\rangle,\ y\rangle\big)\ \big(\text{coin}\ \langle\rangle\big)^{\downarrow} \\
&\longrightarrow^{*} \big(\lambda x.(\lambda y.\langle \text{coin}\ \langle\rangle,\ y\rangle)\big(\dots\big((\lambda y.\langle \text{coin}\ \langle\rangle,\ y\rangle)\ x\big)\dots\big)\big)\ \big(\text{coin}\ \langle\rangle\big)^{\downarrow} \\
&\longrightarrow (\lambda y.\langle \text{coin}\ \langle\rangle,\ y\rangle)\ \big(\dots\ \big((\lambda y.\langle \text{coin}\ \langle\rangle,\ y\rangle)\ (\text{coin}\ \langle\rangle)^{\downarrow}\big)\dots\big) \\
&\longrightarrow^{*} \big\langle \underbrace{(\text{coin}\ \langle\rangle)^{\downarrow},\dots,(\text{coin}\ \langle\rangle)^{\downarrow}}_{n\ \text{times}} \big\rangle
\end{aligned}
$$

Simply typed case:

We cannot derive a type for t_{toss} (or a similar function term) in simply typed *QLC*.

Polymorphically typed case:

After appropriate modifications of function term t_{toss}, we get function term t_{poly} which includes proper type annotations (see Example 4.75 for details). This leads to the following full evaluation.

$$
\begin{aligned}
t_{\text{poly}} &\equiv \overbrace{\big(\Lambda X{<:}!Top.\lambda f{:}!\Phi_f.\lambda x{:}!X.f\ X^{!\otimes n-1}\ (f\ X^{!\otimes n-2}\ (\dots(f\ !X\ x)\dots))\big)}^{\equiv\ulcorner n-1\urcorner^{!\otimes}}\ !Bit\ \big(\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle\big)\ \big(\text{coin}_{\text{poly}}\ \langle\rangle\big) \\
&\longrightarrow^{*} \big(\lambda f{:}!\Phi_f[!Bit/X].\lambda x{:}!!Bit.f\ (!Bit)^{!\otimes n-1}\ (f\ (!Bit)^{!\otimes n-2}(\dots(f\ !!Bit\ x)\dots))\big)\ \big(\Lambda Y{<:}!Top.\lambda y{:}!Y.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle\big)\ \big(\text{coin}_{\text{poly}}\ \langle\rangle\big)^{\downarrow} \\
&\longrightarrow^{*} \big(\lambda x{:}!!Bit.\big((\lambda y{:}(!Bit)^{!\otimes n-1}.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle\big)\ \big((\lambda y{:}(!Bit)^{!\otimes n-2}.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle\big)\ \big(\dots\big((\lambda y{:}!!!Bit.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle\big)\ x\big)\dots\big)\big)\big)\big)\ \big(\text{coin}_{\text{poly}}\ \langle\rangle\big)^{\downarrow} \\
&\longrightarrow (\lambda y{:}(!Bit)^{!\otimes n-1}.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle)\ \big((\lambda y{:}(!Bit)^{!\otimes n-2}.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle)\ \big(\dots\ \big((\lambda y{:}!!!Bit.\langle \text{coin}_{\text{poly}}\ \langle\rangle,\ y\rangle)\ (\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}\big)\dots\big)\big) \\
&\longrightarrow^{*} \big\langle \underbrace{(\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow},\dots,(\text{coin}_{\text{poly}}\ \langle\rangle)^{\downarrow}}_{n\ \text{times}} \big\rangle
\end{aligned}
$$

[59] For the sake of a simpler presentation, we neglect to write down the full quantum closures and the associated reduction probabilities here and also in the subsequent evaluation, although they are actually a formal requirement.

C.2 Type derivation for a non-terminating polymorphic function term

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash f{:}!(\mathit{Unit} \multimap \Psi), x{:}\mathit{Unit}}{\vdash (\mathit{Unit} \multimap \Psi)}}{\vdash\ !(\mathit{Unit} \multimap \Psi) \prec: (\mathit{Unit} \multimap \Psi)}}{f{:}!(\mathit{Unit} \multimap \Psi), x{:}\mathit{Unit} \vdash f : (\mathit{Unit} \multimap \Psi)}\,(\mathrm{tmVar}) \quad \dfrac{\vdash \emptyset}{\vdash \langle\rangle : \mathit{Unit}}}{f{:}!(\mathit{Unit} \multimap \Psi), x{:}\mathit{Unit} \vdash (f\ \langle\rangle) : \Psi} \quad \mathit{ftmv}\big((f\ \langle\rangle)\big) \cap |x{:}\mathit{Unit}| = \emptyset}{f{:}!(\mathit{Unit} \multimap \Psi) \vdash (\lambda x{:}\mathit{Unit}.f\ \langle\rangle) : !(\mathit{Unit} \multimap \Psi)}\,(\mathrm{nlTmAbs}) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash f{:}!(\mathit{Unit} \multimap \Psi)}{\vdash (\mathit{Unit} \multimap \Psi)}}{\vdash\ !(\mathit{Unit} \multimap \Psi) \prec: (\mathit{Unit} \multimap \Psi)}}{f{:}!(\mathit{Unit} \multimap \Psi) \vdash f : (\mathit{Unit} \multimap \Psi)}\,(\mathrm{tmVar}) \quad \dfrac{\vdash \emptyset}{\vdash \langle\rangle : \mathit{Unit}}}{f{:}!(\mathit{Unit} \multimap \Psi) \vdash (f\ \langle\rangle) : \Psi}\,(\mathrm{tmApp})}{\vdash (\mathit{letrec}\ f{:}!(\mathit{Unit} \multimap \Psi) = (\lambda x{:}\mathit{Unit}.f\ \langle\rangle)\ \mathit{in}\ (f\ \langle\rangle)) : \Psi}\,(\mathrm{rec})$$

D Collection of all relevant derivation rules

Defined derivation rules

Proved types:

$$\frac{\vdash \Theta}{\Theta \vdash !^n \mathit{Top}} \text{ (}\mathit{Top}\text{ type)} \qquad \frac{\vdash \Theta}{\Theta \vdash !^n \mathit{Unit}} \text{ (}\mathit{Unit}\text{ type)} \qquad \frac{\vdash \Theta}{\Theta \vdash \mathit{Qbit}} \text{ (}\mathit{Qbit}\text{ type)}$$

$$\frac{\vdash \Theta, X{<:}\Phi_X, \Theta'}{\Theta, X{<:}\Phi_X, \Theta' \vdash X} \text{ (linear type variable)} \qquad \frac{\Theta \vdash \Phi \qquad \Theta \vdash \Psi}{\Theta \vdash !^n(\Phi \multimap \Psi)} \text{ (function type)}$$

$$\frac{\vdash \Theta, X{<:}!\Phi_X, \Theta'}{\Theta, X{<:}!\Phi_X, \Theta' \vdash !^n X} \text{ (nonlinear type variable)}$$

$$\frac{\Theta \vdash !^n\Phi \qquad \Theta \vdash !^n\Psi}{\Theta \vdash !^n(\Phi \otimes \Psi)} \text{ (product type)} \qquad \frac{\Theta \vdash !^n\Phi \qquad \Theta \vdash !^n\Psi}{\Theta \vdash !^n(\Phi \oplus \Psi)} \text{ (sum type)}$$

$$\frac{\vdash \Theta, \Theta' \qquad \Theta, X{<:}\Phi_X, \Theta' \vdash \Psi \qquad X \notin \mathit{nftyv}(\Psi)}{\Theta, \Theta' \vdash (\forall X{<:}\Phi_X.\Psi)} \text{ (linear-polymorphic type)}$$

$$\frac{\Theta, \Theta' \vdash \Phi_X <: !\mathit{Top} \qquad \Theta, X{<:}\Phi_X, \Theta' \vdash !^n\Psi}{\Theta, \Theta' \vdash !^n(\forall X{<:}\Phi_X.\Psi)} \text{ (nonlinear-polymorphic type)}$$

Proved subtypes:

$$\frac{\Theta \vdash \Phi}{\Theta \vdash \Phi <: \mathit{Top}} \text{ (}\mathit{Top}\text{ supertype)} \qquad \frac{\Theta \vdash \Phi}{\Theta \vdash \Phi <: \Phi} \text{ (}<:\text{ reflexivity)}$$

$$\frac{\Theta \vdash \Phi <: \Psi}{\Theta \vdash !\Phi <: \Psi} \text{ (! left)} \qquad \frac{\Theta \vdash !\Phi <: \Psi}{\Theta \vdash !\Phi <: !\Psi} \text{ (! right)}$$

$$\frac{\Theta \vdash \Phi <: \Phi' \qquad \Theta \vdash \Psi <: \Psi'}{\Theta \vdash (\Phi' \multimap \Psi) <: (\Phi \multimap \Psi')} \text{ (function subtype)}$$

$$\frac{\Theta \vdash \Phi_1 <: \Psi_1 \qquad \Theta \vdash \Phi_2 <: \Psi_2}{\Theta \vdash (\Phi_1 \otimes \Phi_2) <: (\Psi_1 \otimes \Psi_2)} \text{ (product subtype)}$$

$$\frac{\Theta \vdash \Phi_1 <: \Psi_1 \qquad \Theta \vdash \Phi_2 <: \Psi_2}{\Theta \vdash (\Phi_1 \oplus \Phi_2) <: (\Psi_1 \oplus \Psi_2)} \text{ (sum subtype)}$$

$$\frac{\Theta, \Theta' \vdash \Phi_X <: \Phi'_X \qquad \Theta, X{<:}\Phi_X, \Theta' \vdash \Psi <: \Psi'}{\Theta, \Theta' \vdash (\forall X{<:}\Phi'_X.\Psi) <: (\forall X{<:}\Phi_X.\Psi')} \text{ (polymorphic subtype)}$$

Proved terms:

$$\frac{\Theta \vdash \Gamma \qquad \Theta \vdash \Psi \qquad \Theta \vdash !(Bit \multimap Qbit) \prec: \Psi}{\Theta|\Gamma \vdash \text{new} : \Psi}\ \text{(new operator)} \qquad \frac{\Theta \vdash \Gamma \qquad \Theta \vdash \Psi \qquad \Theta \vdash !(Qbit \multimap !Bit) \prec: \Psi}{\Theta|\Gamma \vdash \text{meas} : \Psi}\ \text{(meas operator)}$$

$$\frac{\Theta \vdash \Gamma \qquad \Theta \vdash \Psi \qquad \Theta \vdash !(Qbit^{\otimes m+1} \multimap Qbit^{\otimes m+1}) \prec: \Psi \qquad U \in \mathcal{U}_{m+1}}{\Theta|\Gamma \vdash U : \Psi}\ \text{(unitary operator)}$$

$$\frac{\Theta \vdash \Gamma, x{:}\Phi_x \qquad \Theta \vdash \Psi_x \qquad \Theta \vdash \Phi_x \prec: \Psi_x}{\Theta|\Gamma, x{:}\Phi_x \vdash x : \Psi_x}\ \text{(term variable)} \qquad \frac{\Theta|\Gamma, x{:}\Phi_x \vdash t : \Psi}{\Theta|\Gamma \vdash (\lambda x{:}\Phi_x.t) : (\Phi_x \multimap \Psi)}\ \text{(linear term abstraction)}$$

$$\frac{\Theta|!\Delta, \Gamma, x{:}\Phi_x \vdash t : \Psi \qquad ftmv(t) \cap |\Gamma| = \emptyset}{\Theta|!\Delta, \Gamma \vdash (\lambda x{:}\Phi_x.t) : !^{n+1}(\Phi_x \multimap \Psi)}\ \text{(nonlinear term abstraction)}$$

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash s : !^n(\Phi \multimap \Psi) \qquad \Theta|!\Delta, \Gamma_2 \vdash t : \Phi}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash (s\ t) : \Psi}\ \text{(term application)}$$

$$\frac{\vdash \Theta, \Theta' \qquad \Theta, X{<:}\Phi_X, \Theta'|\Gamma \vdash t : \Psi \qquad X \notin ftyv(\Gamma) \qquad X \notin nftyv(\Psi)}{\Theta, \Theta'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : (\forall X{<:}\Phi_X.\Psi)}\ \text{(linear type abstraction)}$$

$$\frac{\Theta, \Theta' \vdash \Phi_X <: !Top \qquad \Theta, X{<:}\Phi_X, \Theta'|\Gamma \vdash t : !^n\Psi \qquad X \notin ftyv(\Gamma)}{\Theta, \Theta'|\Gamma \vdash (\Lambda X{<:}\Phi_X.t) : !^n(\forall X{<:}\Phi_X.\Psi)}\ \text{(nonlinear type abstraction)}$$

$$\frac{\Theta|\Gamma \vdash t : !^n(\forall X{<:}\Phi_X.\Psi) \qquad \Theta \vdash \Upsilon \qquad \Theta \vdash \Upsilon <: \Phi_X}{\Theta|\Gamma \vdash (t\ \Upsilon) : !^n\Psi[\Upsilon/X]}\ \text{(type application)}$$

$$\frac{\Theta \vdash \Gamma}{\Theta|\Gamma \vdash \langle\rangle : !^n\,Unit}\ (Unit\ \text{term}) \qquad \frac{\Theta|!\Delta, \Gamma_1 \vdash t_1 : !^n\Phi \qquad \Theta|!\Delta, \Gamma_2 \vdash t_2 : !^n\Psi}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash \langle t_1, t_2\rangle : !^n(\Phi \otimes \Psi)}\ \text{(pair term)}$$

$$\frac{\Theta|\Gamma, x{:}!^n\Phi_x, y{:}!^n\Phi_y \vdash t : \Psi}{\Theta|\Gamma \vdash (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t) : (!^n(\Phi_x \otimes \Phi_y) \multimap \Psi)}\ \text{(linear pair abstraction)}$$

$$\frac{\Theta|!\Delta, \Gamma, x{:}!^n\Phi_x, y{:}!^n\Phi_y \vdash t : \Psi \qquad ftmv(t) \cap |\Gamma| = \emptyset}{\Theta|!\Delta, \Gamma \vdash (\lambda\langle x{:}\Phi_x, y{:}\Phi_y\rangle.t) : !^m(!^n(\Phi_x \otimes \Phi_y) \multimap \Psi)}\ \text{(nonlinear pair abstraction)}$$

$$\frac{\Theta|\Gamma \vdash t : !^n\Phi_l \qquad \Theta \vdash !^n\Phi_r}{\Theta|\Gamma \vdash inj_l(t) : !^n(\Phi_l \oplus \Phi_r)}\ \text{(left injection)} \qquad \frac{\Theta \vdash !^n\Phi_l \qquad \Theta|\Gamma \vdash t : !^n\Phi_r}{\Theta|\Gamma \vdash inj_r(t) : !^n(\Phi_l \oplus \Phi_r)}\ \text{(right injection)}$$

$$\frac{\Theta|!\Delta, \Gamma_1 \vdash s : !^n(\Phi_x \oplus \Phi_y) \qquad \Theta|!\Delta, \Gamma_2 \vdash (\lambda x{:}\Phi_x.t_l) : !^m(!^n\Phi_x \multimap \Psi) \qquad \Theta|!\Delta, \Gamma_2 \vdash (\lambda y{:}\Phi_y.t_r) : !^m(!^n\Phi_y \multimap \Psi)}{\Theta|!\Delta, \Gamma_1, \Gamma_2 \vdash \big(match\ s\ with\ (\lambda x{:}\Phi_x.t_l) \mid (\lambda y{:}\Phi_y.t_r)\big) : \Psi}\ \text{(case distinction)}$$

$$\frac{\Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi) \vdash (\lambda x{:}\Phi_x.s) : !(\Phi_x \multimap \Psi) \qquad \Theta|!\Delta, f{:}!(\Phi_x \multimap \Psi), \Gamma \vdash t : \Upsilon}{\Theta|!\Delta, \Gamma \vdash \big(letrec\ f{:}!(\Phi_x \multimap \Psi) = (\lambda x{:}\Phi_x.s)\ in\ t\big) : \Upsilon}\ \text{(recursion)}$$

Derived derivation rules

Proved types:

$$\frac{\vdash \widehat{\Theta} \qquad \Theta \vdash \Phi \qquad \Theta \sqsubseteq \widehat{\Theta}}{\widehat{\Theta} \vdash \Phi}\ \text{(type weakening)} \qquad \frac{\vdash \Theta^{\pi} \qquad \Theta \vdash \Phi}{\Theta^{\pi} \vdash \Phi}\ \text{(type permutation)}$$

$$\frac{\Theta \vdash !^{m}\Upsilon \qquad \Theta, Y{<:}!^{n}\Phi_Y, \Theta' \vdash \Psi \qquad \begin{array}{c}\Phi_Y, \Upsilon \text{ linear} \\ n > 0 \Rightarrow m > 0\end{array}}{\Theta, Y{<:}!^{m}\Upsilon, \Theta' \vdash \Psi}\ \text{(type bound replacement)}$$

$$\frac{\Theta \vdash !\Phi}{\Theta \vdash !^{m+1}\Phi}\ \text{(! amplification)} \qquad \frac{\Theta \vdash !^{n+1}\Phi}{\Theta \vdash \Phi}\ \text{(! elimination)}$$

$$\frac{\begin{array}{ccc}\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi & \Theta \vdash \Upsilon & X \notin \mathit{nftyv}(\Theta') \\ & \Theta \vdash \Upsilon <: \Phi_X & X \notin \mathit{nftyv}(\Psi)\end{array}}{\Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X]}\ \text{(linear type substitution)}$$

$$\frac{\Theta, X{<:}\Phi_X, \Theta' \vdash \Psi \qquad \begin{array}{c}\Theta \vdash !\Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array}}{\Theta, \Theta'[\Upsilon/X] \vdash \Psi[\Upsilon/X]}\ \text{(nonlinear type substitution)}$$

$$\frac{\Theta \vdash !^{n}(\forall X{<:}\Phi_X.\Psi) \qquad \begin{array}{c}\Theta \vdash \Upsilon \\ \Theta \vdash \Upsilon <: \Phi_X\end{array}}{\Theta \vdash !^{n}\Psi[\Upsilon/X]}\ \text{(bounded type substitution)}$$

Proved subtypes:

$$\frac{\vdash \widehat{\Theta} \qquad \Theta \vdash \Phi <: \Psi \qquad \Theta \sqsubseteq \widehat{\Theta}}{\widehat{\Theta} \vdash \Phi <: \Psi}\ \text{(subtype weakening)} \qquad \frac{\vdash \Theta^{\pi} \qquad \Theta \vdash \Phi <: \Psi}{\Theta^{\pi} \vdash \Phi <: \Psi}\ \text{(subtype permutation)}$$

Proved terms:

$$\frac{\widehat{\Theta} \vdash \widehat{\Gamma} \qquad \Theta|\Gamma \vdash t : \Phi \qquad \begin{array}{c}\Theta \sqsubseteq \widehat{\Theta} \\ \Gamma \subseteq \widehat{\Gamma}\end{array}}{\widehat{\Theta}|\widehat{\Gamma} \vdash t : \Phi}\ \text{(term weakening)} \qquad \frac{\vdash \Theta^{\pi} \qquad \Theta|\Gamma \vdash t : \Phi}{\Theta^{\pi}|\Gamma \vdash t : \Phi}\ \text{(term permutation)}$$

References

[Abr93] Abramsky, S. *Computational Interpretations of Linear Logic.* In: *Theoretical Computer Science* 111 (1993), pp. 3–57

[AD04] Arrighi, P.; Dowek, G. *Linear-algebraic lambda-calculus.* In: *Proceedings of the Second International Workshop on Quantum Programming Languages, QPL 2004.* TUCS General Publication 33, Turku Centre for Computer Science, Turku, Finland. (2004), pp. 21–38

[AD05] Arrighi, P.; Dowek, G. *A computational definition of the notion of vectorial space.* In: *Electronic Notes in Theoretical Computer Science* 117 (2005), pp. 249–261

[AD08] Arrighi, P.; Dowek, G. *Linear-algebraic lambda-calculus: higher-order, encodings, and confluence.* In: *Rewriting Techniques and Applications, RTA 2008.* LNCS 5117, Springer-Verlag, Hagenberg. (2008), pp. 17–31

[ADC11] Arrighi, P.; Díaz-Caro, A. *Scalar System F for Linear-Algebraic λ-Calculus: Towards a Quantum Physical Logic.* In: *Electronic Notes in Theoretical Computer Science* 270(2) (2011), pp. 219–229

[ADV11] Arrighi, P.; Díaz-Caro, A.; Valiron, B. *A Type System for the Vectorial Aspects of the Linear-Algebraic Lambda-Calculus.* To appear in *Proceedings of the Seventh International Workshop on Developments of Computational Models, DCM 2011.* Also available as ArXiv preprint (2011), arXiv:1012.4032v2 [cs.LO]

[Bar92] Barendregt, H.P. *Lambda Calculi with Types.* In: Abramsky, S.; Gabbay, D.M.; Maibaum, T.S.E. (eds.) *Handbook of Logic in Computer Science, Volume II.* Oxford University Press, Oxford, New York. (1992), pp. 117–309

[Bie94] Bierman, G.M. *On Intuitionistic Linear Logic.* Ph.D. thesis, Computer Laboratory, University of Cambridge. 1994

[BV97] Bernstein, E.; Vazirani, U. *Quantum complexity theory.* In: *SIAM Journal of Computing* 26(5) (1997), pp. 1411–1473

[Cro93] Crole, R.L. *Categories for types.* Cambridge University Press, Cambridge. 1993

[CMM91] Cardelli, L.; Martini, S.; Mitchell, J.C.; Scedrov, A. *An extension of system F with subtyping.* In: *International Conference on Theoretical Aspects of Computer Software.* LNCS 526, Springer-Verlag, Berlin. (1991), pp. 750–770

[DAG11] Díaz-Caro, A.; Arrighi, P.; Gadella, M.; Grattage, J. *Measurements and Confluence in Quantum Lambda Calculi With Explicit Qubits.* In: *Electronic Notes in Theoretical Computer Science* 270(1) (2011), pp. 59–74

[Deu85] Deutsch, D. *Quantum theory, the Church-Turing principle and the universal quantum computer.* In: *Proceedings of the Royal Society of London, Series A* 400 (1985), pp. 97–117

[Die82] Dieks, D. *Communication by EPR devices.* In: *Physical Letters A* 92(6) (1982), pp. 271–272

[DMZ09] Dal Lago, U.; Masini, A.; Zorzi, M. *On a measurement-free quantum lambda calculus with classical control.* In: *Mathematical Structures in Computer Science* 19(2) (2009), pp. 297–335

[DMZ11] Dal Lago, U.; Masini, A.; Zorzi, M. *Confluence Results for a Quantum Lambda Calculus with Measurements.* In: *Electronic Notes in Theoretical Computer Science* 270(2) (2011), pp. 251–261

[FD01] Florido, M.; Damas, L. *Intersection Types and the Linear Lambda-Calculus.* Technical Report DCC-2001-9, DCC – FC & LIACC, Universidade do Porto. (2001)

[Gay06] Gay, S. *Quantum Programming Languages – Survey and Bibliography.* In: *Mathematical Structures in Computer Science* 16(4) (2006), pp. 581–600

[Gir87] Girard, J.Y. *Linear Logic.* In: *Theoretical Computer Science* 50(1) (1987), pp. 1–102

[GTL90] Girard, J.Y.; Taylor, P.; Lafont, Y. *Proofs and Types.* Cambridge University Press, Cambridge, New York. 1990

[Gru99] Gruska, J. *Quantum Computing.* McGraw Hill, London. 1999

[Kni96] Knill, E.H. *Conventions for quantum pseudocode.* Technical Report LAUR-96-272, Los Alamos National Laboratory. (1996)

[Mil78] Milner, R. *A Theory of Type Polymorphism in Programming.* In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375

[NC00] Nielsen, M.A.; Chuang, I.L. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, New York. 2000

[NO08] Nakahara, M.; Ohmi, T. *Quantum Computing – From Linear Algebra to Physical Realizations.* CRC Press, Boca Raton, London. 2008

[Pie94] Pierce, B.C. *Bounded Quantification is Undecidable.* In: *Information and Computation* 112(1) (1994), pp. 131–165

[Pie02] Pierce, B.C. *Types and Programming Languages.* The MIT Press, Cambridge, London. 2002

[Sch98] Schubert, A. *Second-order unification and type inference for Church-style polymorphism.* In: *The twenty-fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998.* ACM Press. (1998), pp. 279–288

[Sel04a] Selinger, P. *A Brief Survey on Quantum Programming Languages.* In: *Proceedings of the seventh International Symposium on Functional and Logic Programming, Nara, Japan.* LNCS 2998, Springer-Verlag, Berlin. (2004), pp. 1–6

[Sel04b] Selinger, P. *Towards a quantum programming language.* In: *Mathematical Structures in Computer Science* 14(4) (2004), pp. 527–586

[SU06] Sørensen, M.H.; Urzyczyn, P. *Lectures on the Curry-Howard Isomorphism.* Studies in logic and the foundations of mathematics, volume 149. Elsevier, Amsterdam, Boston. 2006

[SV05] Selinger, P.; Valiron, B. *A lambda calculus for quantum computation with classical control.* In: *Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications, TLCA 2005.* LNCS 3461, Springer-Verlag, Berlin. (2005), pp. 354–368

[SV06] Selinger, P.; Valiron, B. *A lambda calculus for quantum computation with classical control.* In: *Mathematical Structures in Computer Science* 16(3) (2006), pp. 527–552

[SV08a] Selinger, P.; Valiron, B. *A linear-non-linear model for a computational call-by-value lambda calculus.* In: *Proceedings of the Eleventh International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2008.* LNCS 4962, Springer-Verlag, Berlin. (2008), pp. 81–96

[SV08b] Selinger, P.; Valiron, B. *On a fully abstract model for a quantum linear functional language.* In: *Proceedings of the Fourth International Workshop on Quantum Programming Languages, QPL 2006.* Electronic Notes in Theoretical Computer Science 210 (2008), pp. 123–137

[SV09] Selinger, P.; Valiron, B. *Quantum lambda calculus.* In: Gay, S.; Mackie, I. (eds.) *Semantic Techniques in Quantum Computation.* Cambridge University Press. (2009), pp. 135–172

[Val04a] Valiron, B. *A functional programming language for quantum computation with classical control.* Master's thesis, Department of Mathematics and Statistics, University of Ottawa. 2004

[Val04b] Valiron, B. *Quantum typing.* In: *Proceedings of the Second International Workshop on Quantum Programming Languages, QPL 2004.* TUCS General Publication 33, Turku Centre for Computer Science, Turku, Finland. (2004), pp. 163–178

[Val08] Valiron, B. *Semantics for a Higher-Order Functional Programming Language for Quantum Computation.* Ph.D. thesis, Department of Mathematics and Statistics, University of Ottawa. 2008

[Val11] Valiron, B. *On Quantum and Probabilistic Linear Lambda-calculi (Extended Abstract).* In: *Electronic Notes in Theoretical Computer Science* 270(1) (2011), pp. 121–128

[vTo04] van Tonder, A. *A lambda calculus for quantum computation.* In: *SIAM Journal of Computing* 33(5) (2004), pp. 1109–1135

[Wal05] Walker, D. *Substructural Type Systems.* In: Pierce, B.J. (ed.) *Advanced Topics in Types and Programming Languages.* The MIT Press, Cambridge, London. (2005), pp. 3–43

[Wel94] Wells, J.B. *Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable.* In: *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, LICS.* (1994), pp. 176–185

[WZ82] Wootters, W.K.; Zurek, W.H. *A single quantum cannot be cloned.* In: *Nature* 299 (1982), pp. 802–803

[Zor09] Zorzi, M. *Lambda Calculi and Logics for Quantum Computing.* Ph.D. thesis, Dipartimento di Informatica, Università degli Studi di Verona. 2009