Implementing a Quantum Program Verifier (Technical Report)

MINGWEI ZHU

1 INTRODUCTION

Formal verification techniques of quantum programs have been long studied and implemented [5]. For example, CoqQ [8] is built upon Quantum Hoare Logic framework in Coq theorem prover to verify quantum programs against specifications for density matrix semantics. Using path sums and SMT solvers, QBricks [1] verifies quantum programs by reasoning about circuit equivalence with a good degree of automation. Previouslyl we proposed a lightweight automated verification framework Qafny for quantum programs [6]. Qafny uses an entanglement type system to track how qubits are grouped together through the program and is built with support for local reasoning. By targeting Dafny, we may reuse existing theorem and lemmas to solve domain-specific proof obligations. Continuing this existing work, we present the formalized implementation of Qafny as a type-directed translation relation in this paper.

2 OVERVIEW

Qafny specializes quantum states into four categories normal, Hadamard, entangled, and Qft quantum states. Each quantum state has different representation in emitted Dafny program to reduce the verification overhead of irreelevant terms.

2.1 Quantum States

A single-qubit state can be generally written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ using Dirac notations [3]. Similarly, a two-qubit state can be expressed as $|\varphi\rangle = \sum_{i\in[0,3]} \alpha_i |i\rangle$, alternatively $\alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle$, where the probability of measuring i is $|\alpha_i|^2$. In general, an n-qubit quantum state can be expressed as a linear combination of computational basis states, $\{|i\rangle \mid i \in [0, 2^n - 1]\}$. For conciseness, we will sometimes omit the coefficients if there exists a trivial normalizing factor. Two quantum states can be composed by taking tensor product; however, not all multi-qubit states can be decomposed into two separate states. For example, $|00\rangle + |10\rangle$ can be decomposed into $(|0\rangle + |1\rangle) \otimes |0\rangle$, while $|00\rangle + |11\rangle$ is not separable and therefore *entangled*.

Quantum gates are used to evolve a state into another. As an example, a Hadamard operator can transform $|0\rangle$ into $|0\rangle + |1\rangle$. Further, entanglements can be created by applying a *controlled gate* over multiple qubits. A controlled-NOT (CNOT) gate takes two qubits, a control qubit and a target qubit, that essentially acts on the tensor product of both states and for each basis state, flips the basis in the target qubit if the basis in the control qubit is 1. Applying CNOT operator to $1/\sqrt{2}(|0\rangle + |1\rangle) \otimes |0\rangle$ (equivalently, $|00\rangle + |10\rangle$) joins both states into one entangled entity, $1/\sqrt{2}\,|00\rangle + 1/\sqrt{2}\,|11\rangle$.

2.2 Manipulating Qubits in Qafny

It is a folklore [2, 7] to treat an ensemble of qubits (a quantum register) as an array of qubits. We may declare a variable `q` as a quantum register of 10 as a binding `q : qreg[10]`. and take a slice of the register with `q[2..4]` to obtain a reference to two qubits. We refer a *slice* or a *range* (meta-variable r) to a slice of quantum register interchangeably.

Ranges are weakly related to each other that one cannot tell if two ranges are entangled or not easily. We need a notion that groups slices together because quantum operations such as the

Author's address: Mingwei Zhu.

controlled-NOT gate could create entanglements between disjoint slices. In QAFNY, we use the term *locus* for a set of slices (possibly) in entanglement, denoted by $\{r, \ldots\}$[1] Quantum operators such as CNOT may induce entanglement between two qubits which effectively merges two loci, say `q[0..1]` and `p[0..1]`, and into one locus `{ q[0..1], p[0..1] }`

2.3 Quantum State Representation

In general, a pure quantum state can be written as

$$|\psi\rangle = \sum_i \sqrt{\alpha_i}\phi_i\,|b_i\rangle$$

where α_i, ϕ_i, and b_i refers to its probability, phase shift, and basis-ket for each state respectively. In our implementation, quantum states are grouped by loci and categorized into the following four *entanglement types*: norm (`nor`), Hadamard (`had`), entangled (`en`), and quantum Fourier transformed (`qft`) states. Each entanglement type corresponds to different kind of state representations in Dafny.

Example 2.1 (Quantum states representation).

Locus	State	Type	Dafny Repr.
`q[0..2]`	$\|0\rangle \otimes \|1\rangle$	`nor`	`[0, 1]`
	$\|2\rangle$	`en`	([1], [2], _)
`p[0..2]`	$(\|0\rangle + \|1\rangle) \otimes (\|0\rangle - \|1\rangle)$	`had`	`([0, 1], 2)`
			`[1/4, 1/4, 1/4, 1/4]`
	$\|0\rangle - \|1\rangle + \|2\rangle - \|3\rangle$	`en`	`[0, 1, 2, 3]`
			`([0, 1, 0, 1], 2)`

Example 2.1 gives two instances of 2-qubit system. The `nor` type models quantum states without superposition or entanglement which are tensor products of basis kets from $\{|0\rangle, |1\rangle\}$, i.e., $\bigotimes_i \varphi_i$ where $\varphi_i \in \{|0\rangle, |1\rangle\}$. Since there's only one eigenstate possible, its amplitude and phase are trivial, therefore such a state is translated into a sequence of 0's and 1's. A `had` state is a tensor product of (multiple) single-qubit quantum states in superposition, i.e., $\bigotimes_i \varphi_i$ where $\varphi_i \in \{|0\rangle + \phi_i\,|1\rangle\}$. Since every ket in such a state shares equivalent possibility and may only differ in the phase shift, each `had` quantum state is translated into a sequence of phase coefficients associated to each $|1\rangle$. For conciseness, each `nor`- and `had`-typed locus may contain only one range: since no entanglement is possible between ranges in such a locus, a locus using more than one range may be refactored into only one range. Consequently, range and locus will be used interchangeably for single-range loci when there's no ambiguity. `en` represents a state in entanglement which is formed by entangling existing loci and would form a loci containing multiple ranges. Its representation consists of a `seq⟨real⟩` for probability, a phase representation as well as a number of `seq⟨nat⟩` of the same length for each basis-ket. Zipping those basis-ket sequences forms a sequence representing the superposition of the tensor product of basis-kets from each slice in entanglement. For example, `q[0..1]` and `p[0..1]` in Example 2.2 forms a `en` locus referring to quantum state $^1/_{\sqrt{2}}\,|0\rangle\,|0\rangle + {}^1/_{\sqrt{2}}\,|1\rangle\,|1\rangle$. This is translated into a `seq⟨real⟩`, `[1/2, 1/2]` for amplitude and two `nat` sequences `q_ket:=[0, 1]` and `p_ket := [0,1]` respectively so that the zipped sequence `[(0,0), (1,1)]` corresponds to $|0\rangle\,|0\rangle$ and $|1\rangle\,|1\rangle$. Phases are omitted in this case as they are trivially one. Formally, given an n-qubit `en` locus consisting of two ranges, `{q[0..m], q[m..n]}`, its corresponding state is

$$\sum_i \alpha_i\phi_i\,\left|\psi_{i,\texttt{q[0..m]}}\right\rangle \left|\psi_{i,\texttt{q[m..n]}}\right\rangle$$

[1]We may omit the curly brackets when possible for conciseness.

where $\psi_{i,r}$ refers to the ith index in the translated basis-ket sequence for r. Since two ranges are now in entanglement and cannot be separated, we maintain the index correspondence between two ranges throughout the translation.

Example 2.2 (`en`*-loci of multiple ranges).*

Locus	State	Basis-Kets	Probabilities
`p[0..1]`, `q[0..1]`	$(\frac{1}{\sqrt{2}}\|0\rangle\|0\rangle + \frac{1}{\sqrt{2}}\|1\rangle\|1\rangle)$	`{[0, 1], [0, 1]}`	`{[1/2, 1/2]}`
`p[0..1]`, `q[0..2]`	$(\frac{1}{2}\|0\rangle\|2\rangle + \frac{1}{2}\|1\rangle\|3\rangle)$	`{[0, 1], [2, 3]}`	`{[1/2, 1/2]}`

It is worth noticing that both ranges/loci have can have two ways to translate based on its type in the context. This suggests a subtyping relation among `nor`, `had` and `en` which will be discussed later in section 3.2. `qft` is a generalization of `en` that is introduced by quantum Fourier transformation operator which will be introduced in section 4.7.

Representing Phases. Phase is a normalized complex coefficient associated to each basis in superposition. Both `had` and `en` states require a phase representation. In Qafny, phases are specified in three form of Root of Unities (RoU). An Nth root of unity is in form $\omega_N = \exp(\pi i/N)$, therefore a phase coefficient can be described using two natural numbers for its power and order. For example, $\omega_n^0 = 1$, $\omega_2^1 = -1$, $\omega_4^1 = i$, and $\omega_4^3 = -i$.

In our system, the phase coefficients of each quantum state can be one of three different translations decided its *phase degree*. The semantics ($[\![\cdot]\!]_{\mathsf{p}}$) of phase type is defined as a function from phase degree to the type of its representation in Dafny. A zeroth-degree phase refers to the trivial case where all phase coefficients are the same. A first-degree phase is translated into a `nat` for order of RoU and a `seq⟨nat⟩` standing for the power of the phase coefficient for each ket in superposition. A second-degree one is similar but used in the case where the coefficient is a sum of root of unities and is carried by `seq⟨nat⟩`.

$$
\begin{array}{lll}
[\![0]\!]_{\mathsf{p}} = \mathtt{unit} & [\![\mathtt{nor}]\!] = \mathtt{seq\langle nat\rangle} & [\![\mathtt{had}]\!] = [\![1]\!]_{\mathsf{p}} \\
[\![1]\!]_{\mathsf{p}} = \mathtt{seq\langle nat\rangle} \times \mathtt{nat} & [\![\mathtt{en}_{n,p}]\!] = \mathtt{seq\langle nat\rangle} \times [\![p]\!]_{\mathsf{p}} \times \mathtt{seq\langle nat\rangle}^n & \\
[\![2]\!]_{\mathsf{p}} = \mathtt{seq\langle seq\langle nat\rangle\rangle} \times \mathtt{nat} & [\![\mathtt{qft}_{n,p}]\!] = \mathtt{seq\langle seq\langle nat\rangle\rangle} \times \mathtt{seq\langle nat\rangle} \times [\![p+1]\!]_{\mathsf{p}} \times \mathtt{seq\langle seq\langle nat\rangle\rangle}^n &
\end{array}
$$

Fig. 1. Semantics of entanglement types

Semantics of Entanglement Types. Figure 1 presents the formal semantics of both locus entanglement type ($[\![\cdot]\!]$) and phase types w.r.t its degree. In particular, `en` is tagged by a phase degree p and the number of slices in the locus n. We will omit the number of slices and the phase degree when there is no ambiguity or when those two parameters are irrelevant.

2.4 Quantum Operations

Here, we give an overview on constructs we use to operate over quantum states.

Application Statements and Operators. An application statement `l *= e` applies to a (sub-)locus an oracle function or a built-in operator such as `H` (Hadamard operator) and `qft` (Quantum Fourier Transform). When an oracle function ($\mu(\overline{\mathtt{x}} \Rightarrow \overline{\mathtt{e}})$) is applied to a list of ranges $\overline{r}$, it binds every basis state in each range r on the LHS to x and execute the corresponding e.

Example 2.3 (Flip Oracle). With `q[0..1], p[0..1]` = $|00\rangle + |10\rangle$, the statement

$$\mathtt{p[0..1]\ {*}{=}\ } \mu(\mathtt{x} \Rightarrow \mathtt{(x\ +\ 1)})^{2}$$

[2] `(x + 1)` is implicitly translated into `(x + 1) % 2` since there's only one qubit in the slice.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Statements}} \\
s & ::= & \texttt{var } x \texttt{:= } e \\
 & \mid & l \texttt{ *= } e \\
 & \mid & \texttt{if } e \texttt{ \{ } s \texttt{ \}} \\
 & \mid & \texttt{for}_{\overline{e}}\ x \in \mathit{int} \texttt{ with } e \texttt{ \{ s \}} \\
 & \mid & s ; s \\
\multicolumn{3}{l}{\text{Locus}} \\
l & ::= & \overline{r} \\
r & ::= & x\texttt{[}e\texttt{..}e\texttt{]} \\
\mathit{int} & ::= & \texttt{[e..e]}
\end{array}
\qquad
\begin{array}{lll}
\multicolumn{3}{l}{\text{Expressions}} \\
e & ::= & x \mid v \\
 & \mid & e \ \mathit{bop} \ e \\
 & \mid & \forall x :: e \\
 & \mid & \texttt{measure r} \\
 & \mid & \mu(\texttt{e} \Rightarrow \texttt{e}) \mid \texttt{qft} \mid \texttt{H} \\
 & \mid & e_{\text{phase}} \mid e_{\text{amp}} \mid e_{\text{pred}} \\
e_{\text{phase}} & ::= & \mathbb{1} \mid \omega(e, e) \mid \Omega\, x \in \texttt{[}e\texttt{..}e\texttt{]} \,.\, \omega(e, e) \\
e_{\text{amp}} & ::= & \texttt{isqrt}(e, e) \mid \texttt{sin}(e) \mid \texttt{cos}(e) \\
v & ::= & n \in \mathbb{N}
\end{array}
$$

Fig. 2. Syntax of Qafny

$$
\begin{array}{lll@{\qquad}l}
e_{\text{pred}} & ::= & \texttt{\{l: } \tau \mapsto \ \_ \texttt{ \}} & (\mathit{wildcard}) \\
 & \mid & \texttt{\{l: nor } \mapsto \otimes\texttt{i. } \texttt{e}_{01}\texttt{\}} & \texttt{e}_{01} := 0 \mid 1 \\
 & \mid & \texttt{\{l: had } \mapsto \otimes\texttt{i. } \texttt{e}_{\text{phase}}\texttt{\}} & \\
 & \mid & \texttt{\{l: en } \mapsto \Sigma \texttt{ i}\in\texttt{[e}_l\texttt{..e}_r\texttt{]. } \texttt{e}_{\text{amp}}\texttt{, } \texttt{e}_{\text{phase}}\texttt{, } (\overline{\texttt{e}_{\text{ket}}})\texttt{\}} & \texttt{e}_{\text{ket}} \in \mathbb{N}
\end{array}
$$

Fig. 3. State predicates

effectively flips the ket in the slice `q[0..1]` so that the state ends up with

$$\{\texttt{q[0..1]}, \texttt{p[0..1]}\} = |01\rangle + |11\rangle.$$

In example 2.3, the corresponding two Dafny sequences `[0, 1], [0, 0]` will be transformed into `[0, 1], [1, 1]` because only the range `p[0..1]` is flipped.
Oracle functions can also define transformations over phases ($\mu(\omega(\texttt{x, n}) \Rightarrow \texttt{e})$).

Generalize Controlled Operations. Controlled operators are generalized to `if` and `for` statements to create entanglement. An `if` statement consists of a guard expression referring to the range used as the control qubits and a body statement that specifies the computation over target qubits.

Example 2.4 (CNOT as an `if` Statement). Using the same initial state in example 2.3, the following program:

```
if q[0..1] { p[0..1] *= μ(x⇒(x+1)); }
```

execute to a program state

$$\{\texttt{q[0..1]}, \texttt{p[0..1]}\} = |00\rangle + |11\rangle\,.$$

In Example 2.4, the oracle operator is only applied to the initial state conditionally: $|10\rangle$ evolves into $|11\rangle$, while $|00\rangle$ stays the same because the basis state of the controlled qubit here is 0 in this case.

State Predicates. Qafny aims to verify if a program behaves w.r.t. the specifications of quantum states. Our program state is a mapping from loci to its quantum state. A predicate over such a program state consists of a locus, its entanglement type, and its quantum state representation in Dafny, denoted by $\texttt{\{l: } \tau \mapsto \texttt{e\}}$. We impose a syntactic restriction to state specification allowed by each type: (1) `nor` loci can only be specified by 0-1 basis; (2) `had` specification only concerns the phase part of the state; (3) `en` specification describes its state per basis states. An amplitude specification is either an inverse square root ($\texttt{isqrt}(e_1, e_2)$) standing for $e_1/\sqrt{e_2}$ or a trigonometric function, `sin` or `cos`. The quantum states from the previous example paragraph can be written as

$$\frac{\begin{array}{c}\Gamma';\emptyset \vdash_{\mathsf{intro}} \overline{e_{req}} : \sigma_{req} \qquad \Gamma';\emptyset \vdash_{\mathsf{intro}} \overline{e_{ens}} : \sigma_{ens} \\ \Gamma' = \Gamma[\overline{x:\kappa}] \qquad \Gamma';\sigma_{req} \vdash s : \sigma_{fwd} \qquad \sigma_{fwd} \le \sigma_{ens}\end{array}}{\Gamma;\sigma \vdash \texttt{method}\ m(\overline{x:\kappa})\ \overline{e_{req}}\ \overline{e_{ens}}\ \{\ s\ \} : \sigma_{ens}}\ \text{[T-MethodDef]}$$

$$\frac{\begin{array}{c} I = \mathrm{dom}(\sigma_I) \qquad \sigma|_{I[x:=e_1]} = \sigma_I[i := e_1] \\ \Gamma;\emptyset \vdash_{\mathsf{intro}} \overline{e_I} : \sigma_I \qquad \Gamma;\sigma_I \vdash S : \sigma_I[i := i+1] \quad \sigma' = (\sigma \setminus I[x := e_1]) \cup \sigma_I[i := e_2]\end{array}}{\Gamma;\sigma \vdash \texttt{for}_{\overline{e_I}}\ \ i\ \in[e_1..e_2]\ \texttt{with}\ b\ \{\ s\ \} : \sigma'}\ \text{[T-For]}$$

$$\frac{\sigma(l) = \mathsf{nor}}{\Gamma;\sigma \vdash l\ \ \texttt{*=}\ \texttt{H} : \sigma[l \mapsto \mathsf{had}]}\ \text{[T-AppHad]} \qquad \frac{l \subseteq l' \quad \sigma(l') = \mathsf{en}}{\Gamma;\sigma \vdash l\ \ \texttt{*=}\ \mu(x \Rightarrow e) : \sigma}\ \text{[T-AppOrcale]}$$

$$\frac{\sigma < \sigma' \quad \Gamma;\sigma' \vdash s : \sigma''}{\Gamma;\sigma \vdash s : \sigma''}\ \text{[T-Sub]} \qquad \frac{\begin{array}{c}\sigma(l_2) = \tau \in \{\mathsf{nor},\mathsf{had}\} \quad l_2 - l_1 = (l_3, l_4) \\ \sigma' = (\sigma \setminus \{l_2\})[l_3 \mapsto \tau, l_3 \mapsto \tau] \quad \Gamma;\sigma' \vdash s : \sigma''\end{array}}{\Gamma;\sigma \vdash s : \sigma''}\ \text{[T-Split]}_3$$

$$\frac{\Gamma[x \mapsto \texttt{qreg[}n\texttt{]}];\sigma[x \mapsto \mathsf{nor}] \vdash s : \sigma'}{\Gamma;\sigma \vdash (\texttt{var}\ \ x\texttt{:=}\ \texttt{nor(}n\texttt{,}\ \ i \Rightarrow e\texttt{);}\ \ s) : \sigma'}\ \text{[T-NewNor]}$$

Fig. 4. Typing rules of Qafny

```
{q[0..1], p[0..1]: en ↦ Σ i∈ [0..2]. isqrt(1, 2), 1, (i, i+1)}.
```

To avoid complicating our system with logical types, state predicates may only be as one standalone assertion.

2.5 Syntax

Qafny is an imperative verification-aware quantum programming DSL which, besides application and `if` statement we have introduces before and standard ones such as declaration and sequencing, features a `for` statement to apply an controlled-unitary operator iteratively over loci. A `for` statement takes in an index variable over a bound interval, a set of invariants, a conditional expression describing the behavior of the control qubits, and loop body statements. The loop body forms a new *frame* that only variables and loci specified in the invariant may be accessed and modified. Among the quantum state passed in the frame, only basis-states in which control qubit condition are satisfied are modified by the body statements. Qafny generates the "boundary conditions" automatically and verifies against the invariant specified. This allows one to reason properties of a local *partial* program states.

3 TYPE RESOLUTION

Entanglement types bundled with each locus is used extensively in the translation process. Our typing relation defined in Figure 4 is a quadruple of a kind environment Γ mapping from program variables to kinds such as `nat` kind or quantum register kinds `qreg`, an entanglement type state σ for locus-type correspondence, the statement s, and a resolved typing state as output.

$$\Gamma;\sigma \vdash s : \sigma'$$

Type information is collected from preconditions and further evolves with the structure of the program to signal when an operation is applicable and whether a type cast (Section 3.2) would be necessary.

3.1 Entanglement Types from Specifications

Since entanglement types are annotated in state predicates, our type resolution gather types from specifications. Type checking and resolution are performed per method following [T-MethodDef]. At the beginning of the method, the resolver initializes the typing state with the locus and types collected from `requires` clauses [T-SpecIntro], in which well-formedness checks are performed to ensure whether loci introduced are indeed pairwise disjoint with each other. Types from assertional constructs such as `assert` statements or `ensures` clauses are checked against typing state as in [T-SpecAssert]. Similar to methods in Dafny, each Qafny method acts as a separate frame that may only modify qubits passed through the binding list.

$$\frac{\begin{array}{c} l \in \mathrm{dom}(\sigma) \lor \sigma(l) = \tau \\ \forall x[e_l\,..\,e_r] \in l.\Gamma(x) = \mathtt{qreg}[e_m] \land e_r \leq e_m \end{array}}{\Gamma;\sigma \vdash_{\mathsf{wf}} l : \tau}\;[\textsc{Wf-Locus}]$$

$$\frac{\Gamma;\sigma \vdash_{\mathsf{wf}} l : \tau}{\Gamma;\sigma \vdash_{\mathsf{intro}} \{l\colon\ \tau \ \mapsto\ e\} : \sigma[l \mapsto \tau]}\;[\textsc{T-SpecIntro}] \qquad \frac{\sigma(l) = \tau}{\Gamma;\sigma \vdash_{\mathsf{assert}} \{l\colon\ \tau \ \mapsto\ e\} : \sigma}\;[\textsc{T-SpecAssert}]$$

$$\frac{e \neq \{\_\colon\ \_ \ \mapsto\ \_\} \quad \Gamma \vdash e : \mathtt{bool}}{\Gamma;\sigma \vdash_{*} e : \sigma}\;[\textsc{T-TrivialSpec}] \qquad \frac{\Gamma;\sigma \vdash_{*} e_1 : \sigma' \qquad \Gamma;\sigma' \vdash_{*} \overline{e} : \sigma''}{\Gamma;\sigma \vdash_{*} e_1, \overline{e} : \sigma''}\;[\textsc{T}^{*}\textsc{-MapM}]$$

Fig. 5. Typing rules for specifications

3.2 Split and Cast

Oracle operators are defined to bind en ranges that belongs to one en locus. As shown in example 2.1, a nor quantum state can be an instance of en state because it is semantically the same as to a singleton en basis. However, applying a flip oracle to a nor state is ill-typed w.r.t. the typing rule [T-AppOracle] because it requires the range to be an en one. We therefore introduces the rules of subtyping ([T-Sub]) for casts in entanglement types with the following subsumption relation.

$$\mathtt{nor} < \mathtt{en} \qquad \mathtt{had} < \mathtt{en}$$

So far, there remains one case that we have not taken care of yet: when creating a new quantum register, the entire register forms a new locus ([T-NewNor]); and, we occasionally need to apply an operation to only a slice of the register and promote its type.

Example 3.1 (Split from nor*).* The following program introduces 10 qubits and apply Hadamard operation to the first qubit identified by `q[0..1]`.

```
var q:= nor(10, _⇒0); q[0..1] *= H;
```

The second statement that is supposed to promote the range `q[0..1]` from nor to had cannot be typed because under the current typing state, only `q[0..10]` is of type nor instead of had. We therefore enrich the type analysis ([T-Split]) to infer the splits from constraints. Now, the prior example first produces two singleton loci, `q[0..1]` and `q[1..10]`, and then applies the Hadamard operator to the first range.

3.3 Invariant Typing

In order to be consistent with the program logics of a loop construct, type resolution must observe the invariant in entanglement types as well. We resolve the entanglement type [T-For] by collecting the entanglement type information from the invariants $\overline{e}_I$ into σ_I and calculating recursively the resulting state σ' from the body s using the invariant typing state. We still need to validate the `for` loop is well-typed on entry. It suffices to check that the restriction of the input typing state σ to all loci I in the invariant typing state σ_I is consistent with σ_I with the index variable set to the interval lower bound. In company with subtyping rule in Section 3.2, a state that may be cast into the initial invariant state can be subsumed. Once checked, the resulting state is constructed by combining the initial state without framed loci with the final frame state from the invariant loop state immediately after the loop terminates.

3.4 Solving Splits in Ranges

The primary efforts in implementing the type resolution with the subtyping relation is to decide if two ranges overlaps. If so, we need to split a slice from an existing range; otherwise, the program may be of wrong type. We currently implement a solver that partially evaluates the difference between two arithmetic expressions w.r.t. the constraints collected from the specification expressions. Our solver can only solve linear integer arithmetic, which turns out to be sufficient in our use case. We prefer this approach because it gives a predicative result and detailed error message on resolution failure. What's more, the partial engine helps later in the translation pass to normalize locus when engineering the translator and reduce trivial expressions that slow down verification when there are too many proof obligations. An alternative is to invoke an SMT solver to decide those interval problems, which we would leave it as a future work.

4 TYPE-DIRECTED TRANSLATION

Type resolution provides entanglement type assignments for all statements on locus basis. Qafny translates program using those type assignments into Dafny statements. Besides those resolved typing state, our translation relation is extended with a state for emitted variables ρ that maps locus in the source language to the emitted variables for quantum state representations in the target language, Dafny:

$$\Gamma, \sigma, \rho \vdash s \rightsquigarrow s' : \sigma' \mid \rho'$$

For presentation purpose, we mark in orange target constructs, asterisk emitted *variables* (asterisked letters should be treated as variables in the target language), and treat all emitted variables not mentioned explicitly to be fresh. Each translation relation is associated to a typing relation, and translation is *fused* with typing resolution. To avoid repetition, each translation rule has a implicit typing judgement $\Gamma, \sigma \vdash s : \sigma'$ in its premise to ensure that the translation is well-typed.

4.1 Translate Locus and State Predicates

An emitted variable state ρ provides query the phase, amplitude/probability, and basis-kets representation of a locus l in the emitted program through $\rho_{\mathsf{phase}}(l)$, $\rho_{\mathsf{amp}}(l)$, and $\rho_r(l)$ respectively where r is the range whose variable of kets we are interested in.

An n-qubit register q can be introduced as a `nor` quantum state with its element constructed by the initializer as specified in [C-NewNor]; the specification over kets in tensor product is translated into a sequence constructor in Dafny. Dually, when translating a predicate, we query the emitted variable for the range ket of interest and transform the binder as a universal quantifier following [C-SpecNor]. Predicates for `en` locus follows the similar fashion but translate every range in the locus as well as amplitudes and phases based on the phase degree associated to the entanglement type as shown in [C-SpecEN].

Phase Predicates. Phase predicates and amplitude predicates are generated separately. Translation of phase predicates depends on phase degree. Zeroth-degree phase doesn't require any verification

$$\frac{e = e[i^*/i] \qquad s_1 = (\texttt{var}\ x^* \texttt{:= seq}\langle\texttt{int}\rangle(n,\ i^* \Rightarrow e)) \qquad \Gamma[x \mapsto \texttt{qreg}[n]];\sigma[x \mapsto \texttt{nor}];\rho[x \mapsto x^*] \vdash s_2 \rightsquigarrow s_2 : \sigma' \mid \rho'}{\Gamma;\sigma \vdash \texttt{var}\ x \texttt{:= nor}(n,\ i \Rightarrow e);s_2 \rightsquigarrow s_1;\ s_2 : \sigma' \mid \rho'}\ [\textsc{C-NewNor}]$$

$$\frac{\Gamma(r) = \texttt{qreg[n]} \qquad r^* = \rho_r(r) \qquad e' = (\forall i^* : \mathsf{nat} \mid i^* < n \cdot r^*[i^*]\ \texttt{==}\ e[i^*/i])}{\Gamma;\sigma;\rho \vdash \{r\texttt{:}\ \ \texttt{nor}\ \mapsto\ \otimes i.\ \ e\} \rightsquigarrow e' : \sigma \mid \rho}\ [\textsc{C-SpecNor}]$$

$$\frac{\begin{array}{c} e_{card} = e_2 - e_1 \qquad \rho \vdash_{\mathsf{amp}} (\overline{r}, e_{amp}) \rightsquigarrow e'_{amp} \qquad \rho \vdash_{\mathsf{phase}} (\overline{r}, p, [e_1..e_2], e_{phase}) \rightsquigarrow e'_{phase} \\ \overline{r^*} = \rho_r(\overline{r}) \quad e'_r = (|r^*|\ \texttt{==}\ e_{card}) \wedge (\forall i^* : \mathsf{nat} \mid i^* < e_{card} \cdot r^*[i^* - e_{card}]\ \texttt{==}\ e_r[i^*/i]) \end{array}}{\Gamma;\sigma;\rho \vdash \{\overline{r}\ :\ \mathsf{en}_p\ \mapsto\ \Sigma i \in [e_1..e_2].\ e_{amp}, e_{phase}, (\overline{e_r})\ \} \rightsquigarrow e'_{amp} \wedge e'_{phase} \wedge \overline{e'_r} : \sigma \mid \rho}\ [\textsc{C-SpecEN}]$$

$$\frac{\overline{r} \subseteq l' \quad r^* = \rho_{\mathsf{ket}}(r) \quad s = (r^*\ \texttt{:= Map}(r^*,\ x \Rightarrow e\ \texttt{\%}\ \texttt{Pow2}(|r|)))}{\Gamma;\sigma;\rho \vdash \overline{r}\ \texttt{*=}\ \mu(\overline{x} \Rightarrow \overline{e}) \rightsquigarrow \overline{s} : \sigma \mid \rho}\ [\textsc{C-AppOracle}]$$

$$\frac{\begin{array}{c} r \in l \qquad l_s \subseteq l \quad \sigma(l) = \mathsf{en} \qquad l' = l \setminus \{r\} \qquad r^* = \rho_{\mathsf{ket}}(r) \\ s_{\mathsf{assert}} = \texttt{assert}\ \forall x^* \texttt{:nat} \mid x^* < e_i \cdot\ r^*[x^*]\ \texttt{==}\ \texttt{0};\ \ \texttt{assert}\ \forall x^* \mid e_i \leq x^* < |r^*| \cdot r^*[x^*]\ \texttt{==}\ 1 \\ s_{\mathsf{split}}{}^* = (r_{\texttt{tt}}{}^*\ \texttt{:=}\ r^*[e_i..];\ r_{\texttt{ff}}{}^*\ \texttt{:=}\ r^*[..e_i]) \qquad (s_{\mathsf{merge}}, \rho') = \mathsf{merge}(\rho_{\texttt{tt}}, \rho_{\texttt{ff}}) \\ \Gamma;\sigma;\rho[r^* := r_{\texttt{tt}}{}^*] \vdash s \rightsquigarrow s_{\texttt{tt}}{}^* : \sigma' \mid \rho_{\texttt{tt}} \qquad \Gamma;\sigma;\rho[r^* := r_{\texttt{ff}}{}^*] \vdash_{\mathsf{noop}} s \rightsquigarrow s_{\texttt{ff}}{}^* : \sigma' \mid \rho_{\texttt{ff}} \end{array}}{\Gamma;\sigma;\rho \vdash \texttt{if}\ r\ \texttt{splitAt}\ e_i\ \{\ s\ \} \rightsquigarrow s_{\mathsf{assert}}; s_{\mathsf{split}}; s_{\texttt{tt}}; s_{\texttt{ff}}; s_{\mathsf{merge}} : \sigma' \mid \rho'}\ [\textsc{C-IfEn}]$$

$$\frac{\begin{array}{c} \sigma(r) = \mathsf{had} \qquad \rho(r) = r^* \\ s_{\mathsf{dup}} = \ r_{\mathsf{dup}}{}^*\ := r^* \qquad \Gamma;\sigma;\rho[r := r_{\mathsf{dup}}{}^*] \vdash s \rightsquigarrow s_{\texttt{tt}} \mid \rho_{\texttt{tt}} \\ (s_{\mathsf{merge}}{}^*, \rho') = \mathsf{merge}(\rho,\ \rho_{\texttt{tt}}) \\ s_r = r_{\mathsf{dup}}{}^*\ \texttt{:= seq}\langle\texttt{nat}\rangle(n,\ \texttt{0})\ \texttt{+}\ \texttt{seq}\langle\texttt{nat}\rangle(n,\ \texttt{1}) \end{array}}{\Gamma;\sigma;\rho \vdash \texttt{if}\ r\ \{\ s\ \} \rightsquigarrow s_{\mathsf{dup}}; s_{\texttt{tt}}; s_r; s_{\mathsf{merge}} : \sigma' \mid \rho'}\ [\textsc{C-IfHad}]$$

Fig. 6. Translation relation from QAFNY to Dafny

by Dafny and translates directly to a tautology ([C-PHASE0]). First-degree phase is similar to the predicate over a ket representation, but it also inserts a check to verify the equality of root ([C-PHASE1]). The phase translation extends naturally to higher-degree cases by qualifying over the extra index parameter in the phase specification. Higher phase degree implies heavier verification overhead and hinders automated reasoning, and we find in practice second degree phase ([C-PHASE2]) is expressive enough: it is only produced by quantum Fourier transformation which are likely to be followed by a measurement construct to reason about the measurement outcomes.

4.2 Translate Operations

We require that all operations applied to a locus to be unitary operations[4] so that it is valid to specific qubits (e.g. a range) in a group of qubits (e.g. a locus) in entanglement. We permit locus where an oracle is applying to a *sub-locus* formed by ranges from the exact locus stored in the

[4]Validating unitary operation is beyond the scope of this paper and left as a future language extension.

$$\frac{}{\rho \vdash_{\mathsf{phase}} (l, 0, int, \mathbb{1}) \rightsquigarrow \texttt{true}}\;[\text{C-Phase0}]$$

$$\frac{\begin{array}{c} e_{card} = e_2 - e_1 \qquad (x_{pow}{}^{*}, x_{root}{}^{*}) = \rho_{\mathsf{phase}}(l) \qquad x_{root}{}^{*} \;\texttt{==}\; e_{root} \\ e'_{phase} \;\texttt{==}\; (|x_{phase}{}^{*}| \;\texttt{==}\; e_{card}) \wedge (\forall i^{*} : \mathsf{nat} \mid i^{*} < e_{card} \cdot x_{phase}{}^{*}\texttt{[}i^{*} - e_{card}\texttt{]} \;\texttt{==}\; e_{pow}\texttt{[}i^{*}\texttt{/}i\texttt{]}) \end{array}}{\rho \vdash_{\mathsf{phase}} (l, 1, \texttt{[}e_1\,\texttt{..}\,e_2\texttt{]}, \omega(e_{pow}, e_{root})) \rightsquigarrow e'_{phase}}\;[\text{C-Phase1}]$$

$$\frac{\begin{array}{c} e_{card} = e_2 - e_1 \qquad e'_{card} = e_r - e_l \qquad (x_{pow}{}^{*}, x_{root}{}^{*}) = \rho_{\mathsf{phase}}(l) \qquad x_{root}{}^{*} \;\texttt{==}\; e_{root} \\ e'_{wf} \;\texttt{==}\; (|x_{phase}{}^{*}| \;\texttt{==}\; e_{card}) \wedge (\forall i^{*} : \mathsf{nat} \mid i^{*} < e_{card} \cdot |x_{phase}{}^{*}\texttt{[}i^{*} - e_{card}\texttt{]}| \;\texttt{==}\; e'_{card}) \\ e'_{phase} \;\texttt{==}\; (\forall i^{*} : \mathsf{nat} \mid \forall x^{*} : \mathsf{nat} \mid i^{*} < e_{card} \cdot x_{phase}{}^{*}\texttt{[}i^{*} - e_{card}\texttt{][}x^{*} - e'_{card}\texttt{]} \;\texttt{==}\; e_{pow}\texttt{[}i^{*}\texttt{/}i\texttt{][}x^{*}\texttt{/}x\texttt{]}) \end{array}}{\rho \vdash_{\mathsf{phase}} (l, 2, \texttt{[}e_1\,\texttt{..}\,e_2\texttt{]}, \Omega\, x \in \texttt{[}e_l\,\texttt{..}\,e_r\texttt{]}.\omega(e_{pow}, e_{root})) \rightsquigarrow e'_{wf} \wedge e'_{phase}}\;[\text{C-Phase2}]$$

Fig. 7. Translation relation of phase specifications

typing state. After having corresponding sequences resolved in the state of emitted variables, we apply a `Map` function with the oracle function provided over those sequences. When translating the oracle definition, its body is implicitly bounded to the size of the matched range to avoid integer overflow.

The number of ranges involved should be consistent with the number of parameters in the oracle function. Entanglement is permitted in the oracle function so that the user may encode a controlled unitary operation rather than using `if` statement to simplify verification in some cases.

Example 4.1 (Translate Flip Oracle). Continuing Example 2.3, let the emitted variable for `p[0..1]` be `p_0_1_en`*. The oracle application is translated into

$$\texttt{p\_0\_1\_en}^{*} \;\texttt{:= Map(x} \Rightarrow \texttt{(x + 1) \% 2, p\_0\_1\_en}^{*}\texttt{);}$$

4.3 Translate Splits and Casts

Splits come frequently with casts. When the type analysis requires both of them, splits are translated before casts because splits can only be performed over `nor` and `had` states while the cast only "promotes" types and invalidates potential split opportunities. [C-SplitNor] presents a translation of split over `nor`-typed locus. Informally, splitting a range r_2 to obtain a subrange r_1 may create two complement sub-ranges r_3 and r_4. The translation process takes slices of the target sequence, emits new variables for matched sub-range, and generate statements to assign sliced sequences to new variables. Splitting a `had` typed state is similar to `nor` with the exception that the variable for phase roots also need to be duplicated.

Example 4.2 (Split and cast). The following method splits the register `q[10]` into two loci, casts the first locus to `en` and applies the oracle to it.

```
method SplitCast(q : qreg[10])
  requires { q[0..10] : nor ↦ ⊗ i . 0 }
  ensures  { q[0..5] : en ↦ ∑ i ∈ [0..1] . 1 }
{
  q[0..5] *= μ(x ⇒ x+1);
}
```

This is translated into

```
method SplitCast(q_0_10_in*:seq<bv1>) returns (p_0_5_out*:seq<nat>);
  requires 10 == |q_0_10_in*|
  requires (∀i :nat | 0≤i<10· q_0_10_in*[i] == 0)
  ensures 1 == |p_0_5_out*|
  ensures (∀i :nat | 0≤i<1· p_0_5_out*[i] == 1)
{
  var q_0_10* :seq<nat> := q_0_10_in*;
  // Declarations ...
  q_0_5* := q_0_10*[0..5];  q_5_10* := q_0_10*[5..10];
  q_5_10_en* := CastNorEn_Ket(q_0_5*);          // Cast
  q_5_10_en* := Map(x => x + 1, q_5_10_en*);    // Oracle Application
  p_0_5_out* := q_5_10_en*;
}
```

$$\frac{\begin{array}{c} r_1 \subseteq r_2 \qquad \sigma(r_2) = \tau \in \{\mathsf{nor}\} \\ r_2 - r_1 = (r_3, r_4) = (x[e_{31}\,..\,e_{32}], x[e_{41}\,..\,e_{42}]) \\ \sigma' = (\sigma \setminus \{r_2\})[r_3 \mapsto \tau, r_3 \mapsto \tau] \\ \Gamma; \sigma' \vdash s : \sigma'' \qquad \Gamma; \sigma'; \rho' \vdash s \rightsquigarrow s' : \sigma'' \mid \rho'' \\ \rho(r_2) = r_2{}^* \qquad \rho' = \rho[r_3 \mapsto_{\mathsf{ket}} r_1{}^*; r_3 \mapsto_{\mathsf{ket}} r_3{}^*; r_4 \mapsto_{\mathsf{ket}} r_4{}^*] \\ e'_3 = e_{32} - e_{31} \qquad e'_2 = e_{41} - e_{31} \qquad e'_4 = e_{42} - e_{31} \\ s_{split} = (r_3{}^* \ \texttt{:=}\ r_2{}^*\texttt{[0..}e'_3\texttt{];}\ r_2{}^* \ \texttt{:=}\ r_2{}^*\texttt{[}e'_3\texttt{..}e'_2\texttt{];}\ r_4{}^* \ \texttt{:=}\ r_2{}^*\texttt{[}e'_2\texttt{..}e'_4\texttt{]}) \end{array}}{\Gamma; \sigma; \rho \vdash s \rightsquigarrow s_{split}\,; s' : \sigma'' \mid \rho''} \ [\textsc{C-SplitNor}]$$

$$\frac{\begin{array}{c} \sigma(r) = \mathsf{nor} \qquad \sigma'(r) = \mathsf{en}_0 \qquad \rho' = \rho[r \mapsto_{\mathsf{ket}} r^*] \\ s_{cast} = (r^* \ \texttt{:=}\ \texttt{CastNorEn\_Ket}(\rho_{\mathsf{ket}}(r))) \qquad \Gamma; \sigma'; \rho' \vdash s \rightsquigarrow s' : \sigma'' \mid \rho'' \end{array}}{\Gamma; \sigma; \rho \vdash s \rightsquigarrow s_{cast}\,; s' : \sigma'' \mid \rho''} \ [\textsc{C-CastNorEn}]$$

$$\frac{\begin{array}{c} \sigma(r) = \mathsf{had} \qquad \sigma'(r) = \mathsf{en}_1 \\ \rho' = \rho[r \mapsto_{\mathsf{ket}} r^*; r \mapsto_{\mathsf{phase}} p^*] \qquad \Gamma; \sigma'; \rho' \vdash s \rightsquigarrow s' : \sigma'' \mid \rho'' \\ s_{cast} = (r^* \ \texttt{:=}\ \texttt{CastHadEn\_Ket}(\rho_{\mathsf{ket}}(r));\ p^* \ \texttt{:=}\ \texttt{CastHadEn\_Phase\_1st}(\rho_{\mathsf{phase}}(r))) \end{array}}{\Gamma; \sigma; \rho \vdash s \rightsquigarrow s_{cast}\,; s' : \sigma'' \mid \rho''} \ [\textsc{C-CastHadEn}]$$

Fig. 8. Translation of Split and Cast

We define two cast translation for nor-en and had-en casts. Since nor doesn't carry nontrivial phase information, it is cast into a 0th degree singleton en state by essentially pack the binary representation into a decimal one. The had case is more involved: for an n-qubit had state, the ket part in the resulting en state spans the basis $\{0 \ldots \texttt{Pow2(n)} - 1\}$ with an equal probability. The phase part is tricky because it requires multiplying and distributing phase coefficient across all resulting $2^n - 1$ terms. We keep the balance between automation and expressiveness by delegating those them to external prelude functions written in Dafny where the phase distribution is defined recursively. Further, we defined specializations to trivial yet common cases, for example, single-qubit had state or had state with trivial phase coefficients. The following function is such an example.

```
function {:opaque} CastNorEn_Ket(q:seq<bv1>) :(c:seq<nat>)
```

```
  requires ∀k :nat | k<|q| · q[k] == 0 || q[k] == 1
  ensures (∀k :nat | k<|q| · q[k] == 0) ==> c == [0]
  ensures c == [LittleEndianNat.ToNatRight(q)]
```

4.4 Translate `if` Conditionals

In general, an `if` statement operates over an `en` state by "zipping" over all range sequences in the locus and execute the body over the rest target ranges conditionally w.r.t. the control qubit. A naive translation is to "filter" over the range sequence that contains control qubit, record the index, and filter the other range sequences in the locus by the index accordingly. This separates the target state into halves and it suffices to execute the body to the sub-state of interest only. This introduces the possibility to reason about a partial state locally. When finishing reasoning about both parts, we merge those two states into through sequence concatenation. In practice, this is far from ideal to the SMT solver because of the layers of indirection introduced by filters in Dafny, and the post-condition is less predictable as the user needs to track the order of elements after filtering, which requires reasoning about the definitional equivalence of quantum states upon some permutation of sequences. Manual proof is inevitable then.

To retain a good degree of automation and keep the proof process easy to follow, we put a simple restriction over the form of the sequences representing the control quantum state: the leading part of the sequence need to evaluate the conditional to `false` while the rest part needs to satisfy it. By introducing an `splitAt` clause to specify the boundary explicitly, we can delimit the partial state easily; this corresponds to the translation rule [C-IfEn]. To ensure the correctness of `splitAt` annotation, we insert a verification time assertion of this partition fact.

Example 4.3 (Translate `if` statements over `en` states). The program

```
if q[0..1] splitAt 2 { p[0..2] *= μ(x⇒(x+1)%4); }
```

is translated into

```
 p_0_2_en_0* := p_0_2_en*[0..2];                    p_0_2_en_1* := p_0_2_en*[2..];
 assert ∀i | 0≤i<|p_0_2_en_0*| · p_0_2_en_0*[i]  == 0;
 assert ∀i | 0≤i<|p_0_2_en_1*| · p_0_2_en_1*[i]  == 1;
 p_0_2_en_1* := Map(x ⇒ (x+1) % 4, p_0_1_en*);  p_0_2_en* := p_0_2_en_0*+p_0_2_en_1*;
```

In this example, `p_0_2_en_0*` holds the slice for $|0\rangle$, acts like applying noop to it, and is finally merged with the other slide `p_0_2_en_1*` on which the unitary gate was applied to.

Another specialized case is where the guard refers to a qubit from a `had` locus. Here we have two approaches to translate this case. The generic approach is to first caset the `had` locus into a `en` one using [C-CastHadEn] by taking the Cartesian product with sequences in the body locus, our experiments show it to be less efficient as it can take long time to verify.

There is a property of `had` type we can exploit. From the typing judgement, it is guaranteed that a `had` locus is disjoint from `en` locus, and a single-qubit `had` state is guaranteed to have only $|0\rangle$ and $|1\rangle$ kets in the translated `en` basis ket. An alternative strategy is to duplicate the state representation for the `en` locus. The `if` body is only applied to one copy and leaves the other one intact. In the end, we only need to concatenate those two copies and create a sequence for the original `had` locus of zeros followed by ones. This avoids the need for specifying the `splitAt` clause or taking a Cartesian product explicitly required by merging two arbitrary `en` sequences.

Example 4.4 (Translate `if` statements over a `had` control). Continuing with the example Example 4.3, let's now assume that `q[0..q]` is instead a `had` range and ignore the `splitAt` clause. The program is then translated using [C-IfHad] into

```
p_0_2_en_0* := p_0_2_en*;    p_0_2_en_1* := p_0_2_en*;
p_0_2_en_1* := Map(x ⇒ (x + 1)  % 2, p_0_1_en*);
p_0_2_en*   := p_0_2_en_0* + p_0_2_en_1*;
n* := |p_0_2_en_0*|;   m* := |p_0_2_en_1*|;
q_0_1_en*   := seq⟨int⟩(n*, _⇒0) + seq⟨int⟩(m*, _⇒1);
```

4.5 Translate for Statements

The translation of `for` statements is similar to the `if` statement, but with a few differences. Consider the program in Example 4.5, which entangles the slice `q[i]` with `p[0..10]` in each iteration. If we translate by unrolling the loop as an `if` statement and following [C-IfHad], the had range will be cast into an en one and added as a *single-qubit* slice into the locus. We end up with a locus $\{$`q[0..1]`$, \dots,$ `q[9..10]`, `p[0..10]`$\}$ which is different from what we are expecting in the invariant: two ranges of 10 qubits, `q[0..10]` and `p[0..10]`. To implement this desired behavior, the `for` statement merges the range acting as the control qubits into an existing slice whose range index is adjacent to it, rather than create a brand-new slice. After joining a had range into an en one on the type level, similar to the translation of the had `if` construct, we generate the program by concatenating the en range sequence to a copy of it by adding `Pow2(`i `-` e_1`)` to each basis. ([C-ForHad])

$$\frac{\begin{array}{c}\sigma_I(r) = \mathsf{had} \quad \Gamma;\sigma;\rho \vdash I \rightsquigarrow I^* \\ s_{\mathsf{guard}}{}^* = r^* \ \mathtt{:=}\ r^* \ \mathtt{+}\ \mathtt{Map}(r^*,\ \_ \Rightarrow x\ \mathtt{+}\ \mathtt{Pow2}(i - e_1)) \\ s_{\mathsf{dup}}{}^* = r_{\mathsf{dup}}{}^* \ \mathtt{:=}\ r^* \quad s_{\mathsf{merge}}{}^*, \rho' = \mathsf{merge}(\rho_{\mathtt{tt}}, \rho_{\mathtt{ff}}) \\ \rho'' = (\rho' \setminus r)[r\mathtt{[i+1/i]} \mapsto r^*] \\ \Gamma;\sigma_I;\rho[r^* \mapsto r_{\mathtt{tt}}{}^*] \vdash s \rightsquigarrow s_{\mathtt{tt}}{}^* \mid \rho_{\mathtt{tt}} \\ \Gamma;\sigma_I;\rho[r^* \mapsto r_{\mathtt{ff}}{}^*] \vdash_{\mathsf{noop}} s \rightsquigarrow s_{\mathtt{ff}}{}^* \mid \rho_{\mathtt{ff}} \\ s_{\mathsf{for}}{}^* = \mathtt{for}_{I^*}\ \mathtt{i}\ \in [e_1 \mathtt{..} e_2]\ s_{\mathsf{split}}{}^*; s_{\mathtt{tt}}{}^*; s_{\mathtt{ff}}{}^*; s_{\mathsf{merge}}{}^*; s_{\mathsf{guard}}{}^* \end{array}}{\Gamma;\sigma_I;\rho \vdash \mathtt{for}_{\overline{e_I}}\ i\ \in [e_1 \mathtt{..} e_2]\ \mathtt{with}\ r\ s \rightsquigarrow s_{\mathsf{for}}{}^* : \sigma' \mid \rho''} \ [\textsc{C-ForHad}]$$

$$\frac{\begin{array}{c} r \in l \quad l_s \subseteq l \quad \sigma_I(l) = \mathsf{en} \quad l' = l \setminus \{r\} \\ s_{\mathsf{split}}{}^* = r_{\mathtt{tt}}{}^* \ \mathtt{:=}\ r^*\mathtt{[}i\mathtt{..]}\ ;\ r_{\mathtt{ff}}{}^* \ \mathtt{:=}\ r^*\mathtt{[..}i\mathtt{]} \\ s_{\mathsf{merge}}{}^*, \rho' = \mathsf{merge}(\rho_{\mathtt{tt}}, \rho_{\mathtt{ff}}) \\ \Gamma;\sigma;\rho \vdash I \rightsquigarrow I^* \\ \Gamma;\sigma_I;\rho[r^* := r_{\mathtt{tt}}{}^*] \vdash s \rightsquigarrow s_{\mathtt{tt}} \mid \rho_{\mathtt{tt}} \\ \Gamma;\sigma_I;\rho[r^* := r_{\mathtt{ff}}{}^*] \vdash_{\mathsf{noop}} s \rightsquigarrow s_{\mathtt{ff}} \mid \rho_{\mathtt{ff}} \\ s_{\mathsf{for}} = \mathtt{for}_{I^*}\ \mathtt{i}\ \in [e_1 \mathtt{..} e_2]\ \{\ s_{\mathsf{split}}; s_{\mathtt{tt}}; s_{\mathtt{ff}}; s_{\mathsf{merge}}\ \} \end{array}}{\Gamma;\sigma_I;\rho \vdash \mathtt{for}_{\overline{e_I}}\ i\ \in [e_1 \mathtt{..} e_2]\ \mathtt{with}\ r\ s \rightsquigarrow s_{\mathsf{for}}{}^* : \sigma' \mid \rho} \ [\textsc{C-ForEn}]$$

Fig. 9. Translation Rules from Qafny to Dafny (Cont.)

Example 4.5 (Modulo Multiplication). `q[1..10]` starts as a had locus.

```
for i in [1..10] with q[i.. i+1]
  invariant {
    q[0..i], p[0..10] : en
    ↦ Σk∈[0..2^i]. (k, Pow(b, k) %10)
  }
```

```
lemma LemmaPowEquiv(s:seq<nat>, a:nat, i:nat, N:nat)
  requires |s| == Pow2(i) && N >= 2
  requires ∀k | 0≤k<Pow2(i) · s[k] == ((Pow(a, Pow2(i)) * (Pow(a, k) % N)) % N)
  ensures ∀k | 0≤k<Pow2(i) · s[k] == (Pow(a, (Pow2(i) + k)) % N)
{
  ∀k | 0≤k<Pow2(i) {
    calc == {
      s[k] == ((Pow(a, Pow2(i)) * (Pow(a, k) % N)) % N);
     { LemmaMulModNoopRightAuto(); }       // crush double Ns
      s[k] == ((Pow(a, Pow2(i)) * Pow(a, k)) % N);
     { LemmaPowAdds(a, Pow2(i), k); }      // crush the add on Power
     s[k] == (Pow(a, (Pow2(i) + k)) % N);
    }
  }
}
```

Fig. 10. Lemma for maintaining loop invariants

```
  invariant { q[i..10] : had ↦ ⊗ k. 1 }
{
  p[0..10] *= μ(x⇒(x*(Pow(b, 2^i))) %10);
}
```

will be translated into

```
for i := 1 to 10
  invariant ...
{
  q_i_10_had*   := q_i_10_had[1..10 − i]*;
  q_0_i_en_0*   := q_0_i_en*;
  p_0_10_en_0*  := p_0_10_en*;
  p_0_10_en*    := Map(x ⇒ (Pow(b, 2^i*x), p_0_10_en*);
  p_0_10_en*    := p_0_10_en_0*+p_0_10_en*;
  q_0_i_en*     := q_0_i_en* + Map(x ⇒ x+2^i, q_0_i_en*);
}
```

In this example, we begin with moving one qubit away from the had range and saving a copy of en sequences to be “modified” into emitted variables ended with _0. The oracle function is then applied to the target sequence to be changed by mapping it over the p_0_10_en*. We then merge the changed part into the unchanged sequence to construct the new p[0]10 and add the removed qubit to q[0]i by adding the exponential.

In our experiment, the loop invariant cannot be proven maintained automatically because of the nature of arithmetic with powers, we instead need to apply a lemma.

```
#LemmaPowEquiv(Repr(p[0 .. 10]), b, i, 10);
```

The leading “#” symbol instructs our translator to treat the statement as an external lemma from Dafny and the Repr keyword is translated directly into the emitted variable for the underlying sequence, i.e., $\rho_{\texttt{p[0..10]}}$, which is p_0_10_en* here. The lemma can be found in Figure 10.

4.6 Translate for statements with en01 type

The following example illustrates the procedure used to create entanglement between n-qubits into a GHZ [4] state.

Example 4.6 (GHZ). Assume that `q[0..1]` is in `en`.

```
for i in [0..n-1] with q[i..i+1] splitAt 1
  invariant {
    q [0..(i+1)] : en ↦ (∑j∈ [0..2]. j * (Pow2(j)-1))
  }
  invariant { q[i+1..n] : nor ↦ ⊗ k . 0 }
{
  q[(i+1)..(i+2)] *= μ(x ⇒ (x+1) % 2);
}
```

For each iteration in Example 4.6, a slice from a `nor` state `q[i+1..i+2]` is entangled with a `en` guard `q[i..i+1]`, which then evolves into a `en` state. In order to prove the invariant under the guard `q[i..i+1]`, we need to split by asserting that each element `e*` after from index 1 in the representation of `q[i..i+1]` must satisfy

`e*/Pow2(i) == 1.`

Proof obligations induced by the specification that verify the binary properties from the decimal representation are usually tricky because it requires knowledge about nonlinear arithmetic. The need for manual proofs becomes inevitable.

We instead provide an alternative binary view of `en`, `en01`, which is translated into Dafny as a `seq⟨seq⟨bv1⟩⟩`. `en01` is almost the same as `en` except the state predicate.

Example 4.7 (GHZ in `en01`). The program states the same as Example 4.6 with only except for the first invariant.

```
{ q [0..(i+1)] : en01 ↦ ∑j∈ [0..2]. ⊗ k ∈ [0.. j]. j }
```

The program is translated into

```
for i := 0to n-1
  invariant 2 ==  |q_0_i1*|
  invariant (∀j |0≤j<2 · (∀k |0≤k<1+i · 1+i ==  |q_0_i1*[j]|))
  invariant (∀j |0≤j<2 · (∀k |0≤k<1+i · q_0_i1*[j][k] == j))
  invariant n-i-1 == |q_i1_n*|
  invariant (∀k |0≤k<n-i-1 · q_i1_n*[k] == 0)
{
  q_0_i1_ff* := q_0_i1*[0..1];  q_0_i1* := q_0_i1*[1..];
  // begin unchanged (casts + guard)
  // split q[i+1..n] into q[i+1..i+2] and q[i+2..n]
  q_i1_i2_ff* := q_i1_n*[0..1];   q_i2_n_ff* := q_i1_n*[1..n-i-1];
  // cast nor into en01 and merge
  q_0_i2_ff* := Map(x5 => x5+q_i1_i2_ff*, q_0_i1_ff*);
  // end unchanged
  // begin changed
  // split q[i+1..n] into q[i+1..i+2] and q[i+2..n]
  q_i1_i2_tt* := q_i1_n*[0..1];   q_i2_n_tt* := q_i1_n*[1..n-i-1];
  // apply the oracle over a single-qubit nor state
```

$$\frac{\begin{array}{c}\{r_x, \overline{r}\} = l \qquad \sigma(l) = \texttt{en} \qquad n = \texttt{Pow2}(|r_x|) \\ \rho' = \rho[l \mapsto_{\texttt{amp}} a^*; l \mapsto_{\texttt{pow}} p^*; l \mapsto_{\overline{r}} \overline{r^*};] \\ s_r = r^* \ := \texttt{seq}\langle\texttt{seq}\langle\texttt{nat}\rangle\rangle(n,\ \_ \Rightarrow \rho_r(l)) \\ s_{r_x} = r_x^* \ := \texttt{seq}\langle\texttt{nat}\rangle(n,\ i^* \Rightarrow i^*) \\ s_{amp} = a^* \ := \texttt{seq}\langle\texttt{seq}\langle\texttt{nat}\rangle\rangle(n,\ \_ \Rightarrow \rho_{\texttt{amp}}(l)) \\ s_{phs} = p^* \ := \texttt{seq}\langle\texttt{seq}\langle\texttt{nat}\rangle\rangle(n, k^* \Rightarrow \texttt{seq}\langle\texttt{nat}\rangle(|\rho_{\texttt{pow}}(l)|, i^* \Rightarrow\ k^* * \rho_{r_x}(l)\texttt{[}i^*\texttt{]+}\ \rho_{\texttt{pow}}(l)\texttt{[}i^*\texttt{]}))\end{array}}{\Gamma; \sigma; \rho \vdash l \ \texttt{*=}\ \texttt{qft} \rightsquigarrow \overline{s_r}; s_{r_x}; s_{phs}; s_{amp} : \sigma[l \mapsto \texttt{qft}] \mid \rho'} \ [\text{C-QftEn}]$$

Fig. 11. Translation relation for QFT

```
19   q_i1_i2_tt* := Map(x=>x+1 % 2, q_i1_i2_tt*[0..1])
20   // cast nor into en01 and merge
21   q_0_i2_tt* := Map(x=>x+q_i1_i2_tt*, q_0_i1*);
22   // end changed
23   // merge changed into unchanged
24   q_0_i2_ff* := q_0_i2_ff*+q_0_i2_tt*;
25   // reassign variables to maintain invariants
26   q_0_i1* := q_0_i2_ff*;   q_i1_n* := q_i2_n_ff*;
27 }
```

en01 can also be understood as a sequence of nor states in superposition. A nor state can be naturally merged into a en01 state by concatenating it to the end of each basis in the en01 state as shown in line 24 and 26 in Example 4.7.

4.7 Translate Quantum Fourier Transform

Quantum Fourier transformation transforms every basis vector in a quantum state into the superposition of kets spanning the entire computational basis with difference in phase coefficient. Applying the transform to a general first degree en n-qubit state has the following effect

$$\sum_{i\in S} \alpha_i \omega(p_i, N)\, |x_i\rangle\, |y_i\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle \left(\sum_{i\in S} \alpha_i \omega(p_i + x_i k, N)\, |y_i\rangle \right)$$

Fitting this structure into an en type is possible but complicates the rest of the verification when reasoning about the outcomes of measuring a specific $|k\rangle$ which is the common case.

Instead, we extend the entanglement type with qft typeFigure 1 that keeps a sequence of basis ket for $|k\rangle$ and quantifies the index over the QFT basis for an inner en state as illustrated in [C-QftEn].

4.8 Translate Methods

Quantum register parameters in a method definition are translated into receivers and producers of its quantum state representation depending on their entanglement type specified in require and ensure clauses. The method body may modify passed quantum registers and is translated into a “pure” method that outputs the resulting state of emitted variables as its return values. Our translator ensures that method invocations follow the same calling convention as the method definition, capturing the resulting state, and reinstate the method frame to the current frame.

4.9 Translate Measurement

Measurement collapses a quantum state into two symbolic variables for the measurement outcome and its probability. The probability is calculated through a Prelude function that computes the sum of probabilities carried in the amplitude representation corresponding the measured ket.

5 LANGUAGE EXTENSIONS

In this section, we present a few useful language extension to simplify the structure and reduce verification overeheads. Those extensions don't interfere with or affect the expressiveness of our existing core QAFNY language.

5.1 Phase Kickback

Recall that had type only contains its phase information and is purely *phaseful.* Although had states are phaseful, it is the common case to apply an non-phaseful oracle over it which require casting it to en first. Unfortunately this generic treatment complicates some trivial properties. For example, applying an oracle $f(x, y) = (x, y + f(x))$ to a had state $\frac{1}{\sqrt{2}} |\phi_n\rangle \otimes (|0\rangle - |1\rangle)$ results in a state $\frac{1}{\sqrt{2}} |\phi_n\rangle \otimes (-1)^{f(x)} (|0\rangle - |1\rangle)$ using a trick called *phase kickback.* In this case, the effect of oracle only adds a $(-1)^{f(x)}$ phase coefficient despite the fact that the oracle itself doesn't mention the phase explicitly. If we are to prove this property by first casting the had locus into en, we would need to reason state equivalence up to some permutation. The reader may find a detail discuss and derivation in Appendix A.

We generalize applying such form of oracle to a en and had locus into a phase kickback language extension: this approach axiomatizes the phase kickback using the fact that had type itself is phaseful: if an oracle operator application involves a had locus, its effect over kets should be treated phasefully. The oracle $f(x, y) = (x, y + f(x))$ is translated using only the phase calculus.

$$\sum_{x\in\phi_n} \frac{1}{\sqrt{2}} \omega_2^{f(x)} |x\rangle |0\rangle + \sum_{x\in\phi_n} \frac{1}{\sqrt{2}} \omega_2^{1-f(x)} |x\rangle |1\rangle$$

Generalizing to arbitrary had, we have

$$\left(\sum_{x\in\phi_n} \alpha_x \omega_N^{k_x} |x\rangle\right) \otimes \frac{1}{\sqrt{2}} \left(\omega_N^0 |0\rangle + \omega_N^k |1\rangle\right) \mapsto \sum_{x\in\phi_n} \frac{1}{\sqrt{2^N}} \omega_N^{k_x + k\cdot f(x)} |x\rangle |0\rangle + \sum_{x\in\phi_n} \frac{1}{\sqrt{2^N}} \omega_N^{k_x + k(1-f(x))} |x\rangle |1\rangle$$

which fits into a first degree en representation.

5.2 Unitary Inversion

Composition of unitary gates without measurements or initialization is reversible. Every pure method defined in Qafny has an adjoint that can "uncompute" the result of it. We extend the language with inverse construct (`inv m`) to the unitary inverse of a method `m`. This in effect flips the pre- and post-condition of the given method.

6 CONCLUSION

This technical report provides a formalized implementation of the core type resolution and translation process. Our lightweight approach stresses automation and verification overhead and is applicable to quantum programs without mixed states that can only be specified within density matrices. An extensive evaluation is under way to demonstrate the expressiveness of our approach which, is beyond the scope of this TR at the current stage. Our current implementation takes about 5000 lines of Haskell code, not counting the AST and its pretty-printer. It involves other experimental extensions, promotion rules between phases of different degrees, and the translation of method that are too verbose to be included in this report.

REFERENCES

[1] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Vol. 12648. Springer International Publishing, Cham, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6 Series Title: Lecture Notes in Computer Science.

[2] Andrew Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. In *APS Meeting Abstracts*.

[3] Paul Adrien Maurice Dirac. 1939. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society* 35 (1939), 416 – 418.

[4] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going beyond Bell's Theorem*. Springer Netherlands, Dordrecht, 69–72. https://doi.org/10.1007/978-94-017-0849-4_10

[5] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. 2023. Formal Verification of Quantum Programs: Theory, Tools, and Challenges. *ACM Transactions on Quantum Computing* 5, 1 (Dec. 2023), 1:1–1:35. https://doi.org/10.1145/3624483

[6] Liyi Li, Mingwei Zhu, Rance Cleaveland, Yi Lee, Le Chang, and Xiaodi Wu. 2024. Qafny: A Quantum-Program Verifier.

[7] Microsoft. 2017. *The Q# Programming Language*. https://docs.microsoft.com/

[8] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. 2023. CoqQ: Foundational Verification of Quantum Programs. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 29:833–29:865. https://doi.org/10.1145/3571222

A DERIVATION OF PHASE KICKBACK

$$\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}|x\rangle|0+f(x)\rangle-\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}|x\rangle|1+f(x)\rangle \tag{1}$$

$$=\begin{bmatrix} f(x)=0\Rightarrow\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}|x\rangle|0\rangle-\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}|x\rangle|1\rangle \\ f(x)=1\Rightarrow\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}|x\rangle|1\rangle-\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}|x\rangle|0\rangle \end{bmatrix} \tag{2}$$

$$=\begin{bmatrix} f(x)=0\Rightarrow\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}(-1)^{f(x)}|x\rangle|0\rangle-\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}(-1)^{f(x)}|x\rangle|1\rangle \\ f(x)=1\Rightarrow-\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}(-1)^{f(x)}|x\rangle|1\rangle+\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}(-1)^{f(x)}|x\rangle|0\rangle \end{bmatrix} \tag{3}$$

$$\equiv\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}(-1)^{f(x)}|x\rangle|0\rangle-\sum_{x\in\phi_n}\frac{1}{\sqrt{2}}(-1)^{f(x)}|x\rangle|1\rangle \tag{4}$$

The main challenging in proving the equation in Qafny resides in reasoning with the ≡ relation which involves the definitional equivalence modulo sequence permutation. A minor wrinkle here is that one also needs to prove by cases over the codomain of $f(x)$ in order to bridge the relation between kets and phases, which is not that easy without axioms between ket representations and phase ones.

A.1 Generalized Phase Kickback

$$U_f\left(\left(\sum_{x\in\phi_n}\alpha_x\omega_N^{k_x}|x\rangle\right)\otimes\frac{1}{\sqrt{2}}\left(\omega_N^0|0\rangle+\omega_N^k|1\rangle\right)\right) \tag{5}$$

$$=\sum_{x\in\phi_n}\alpha_x\omega_N^{k_x}|x\rangle|0+f(x)\rangle+\sum_{x\in\phi_n}\alpha_x\omega_N^{k+k_x}|x\rangle|1+f(x)\rangle \tag{6}$$

$$=\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\begin{bmatrix} f(x)=0\Rightarrow\omega_N^{k_x}|x\rangle|0\rangle+\omega_N^{k+k_x}|x\rangle|1\rangle \\ f(x)=1\Rightarrow\omega_N^{k_x}|x\rangle|1\rangle+\omega_N^{k+k_x}|x\rangle|0\rangle \end{bmatrix} \tag{7}$$

$$\equiv\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\begin{bmatrix} f(x)=0\Rightarrow\omega_N^{k_x}|x\rangle|0\rangle+\omega_N^{k+k_x}|x\rangle|1\rangle \\ f(x)=1\Rightarrow\omega_N^{k+k_x}|x\rangle|0\rangle+\omega_N^{k_x}|x\rangle|1\rangle \end{bmatrix} \tag{8}$$

$$=\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\omega_N^{k_x}\begin{bmatrix} f(x)=0\Rightarrow\omega_N^{0}|x\rangle|0\rangle+\omega_N^{k}|x\rangle|1\rangle \\ f(x)=1\Rightarrow\omega_N^{k}|x\rangle|0\rangle+\omega_N^{0}|x\rangle|1\rangle \end{bmatrix} \tag{9}$$

$$\equiv\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\omega_N^{k_x}\omega_N^{k(0+f(x))}|x\rangle|0\rangle+\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\omega_N^{k_x}\omega_N^{k(1-f(x))}|x\rangle|1\rangle \tag{10}$$

$$=\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\omega_N^{k_x+k\cdot f(x)}|x\rangle|0\rangle+\sum_{x\in\phi_n}\frac{1}{\sqrt{2^N}}\omega_N^{k_x+k(1-f(x))}|x\rangle|1\rangle \tag{11}$$