Runtime Analysis of Quantum Programs
A Formal Approach

Federico Olmedo
Universidad de Chile
Chile

Alejandro Díaz-Caro
Universidad Nacional de Quilmes & ICC
Argentina

The 1st International Workshop on Programming Languages for Quantum Computing
New Orleans, USA – January 2020
The role of quantum program resource analysis

Resource analysis is a particularly (more) relevant problem for quantum programs as it allows:
Resource analysis is a particularly (more) relevant problem for quantum programs as it allows:

1. Validating the “effectivity” of the quantum computing model
Resource analysis is a particularly (more) relevant problem for quantum programs as it allows:

2. Determining which quantum algorithms will be (shortly?) implementable in real quantum hardware
1. Translate the (high-level) program into a (low-level) quantum circuit

```c
#define n 1000
module foo(qbit q[n])
{
    for(int i=0;i<n;i++)
        H(q[i]);
    CNOT(q[n-1],q[0]);
}
module main()
{
    qbit b[n];
    foo(b);
}
```

```c
qbit b[1000];
H ( b[0] );
H ( b[1] );
CNOT ( b[999] , b[0] );
```
Current approach to quantum program resource analysis

1. Translate the (high-level) program into a (low-level) quantum circuit

```qasm
# define n 1000
module foo(qbit q[n])
{
    for(int i=0;i<n;i++)
        H(q[i]);
    CNOT(q[n-1],q[0]);
}
module main()
{
    qbit b[n];
    foo(b);
}
```

2. Read off the **number of qubits and gates** in the circuit

<table>
<thead>
<tr>
<th>Qubit</th>
<th>X</th>
<th>Z</th>
<th>H</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>400</td>
<td>27800</td>
<td>54300</td>
<td>55100</td>
</tr>
</tbody>
</table>
Current approach to quantum program resource analysis

Severe limitations
Current approach to quantum program resource analysis

Severe limitations
Current approach to quantum program resource analysis

Severe limitations
Current approach to quantum program resource analysis

Severe limitations

Same number of gates but (very!) different resource profiles
Current approach to quantum program resource analysis

Severe limitations

Number of gates is a very poor resource measurement
(*likelihood of execution paths is disregarded*)

Same number of gates but (very!) different resource profiles
Severe limitations

Number of gates is a very poor resource measurement (likelihood of execution paths is disregarded)
Current approach to quantum program resource analysis

Severe limitations

Number of gates is a very poor resource measurement (likelihood of execution paths is disregarded)

Restricted to programs with statically bounded loops
Current approach to quantum program resource analysis

Severe limitations

- Number of gates is a very poor resource measurement (likelihood of execution paths is disregarded)
- Restricted to programs with statically bounded loops

```c
#define n 1000
module foo(qbit q[n])
{
    for(int i=0;i<n;i++)
        H(q[i]);
    CNOT(q[n-1],q[0]);
}
```
Current approach to quantum program resource analysis

Severe limitations

Number of gates is a very poor resource measurement
(likelihood of execution paths is disregarded)

Restricted to programs with statically bounded loops

```cpp
#define n 1000
module foo(qbit q[n])
{
    for(int i=0;i<n;i++)
        ...
}
```
Current approach to quantum program resource analysis

Severe limitations

- Number of gates is a very poor resource measurement (likelihood of execution paths is disregarded)

- Restricted to programs with statically bounded loops

Sample algorithms out of scope

- BB84 quantum key distribution algorithm
- Simon’s algorithm

```c
#define n 1000
module foo(qbit q[n])
{
    for(int i=0; i<n; i++)
        ...
    prob. < 1
}
```
Severe limitations

- Number of gates is a very poor resource measurement (*likelihood of execution paths is disregarded*)

- Restricted to programs with statically bounded loops

Sample algorithms out of scope
  - BB84 quantum key distribution algorithm
  - Simon’s algorithm
Severe limitations

Number of gates is a very poor resource measurement (likelihood of execution paths is disregarded)

Restricted to programs with statically bounded loops

Sample algorithms out of scope
  - BB84 quantum key distribution algorithm
  - Simon’s algorithm

Rigid and low-level cost model
Our contribution

Calculus à la weakest precondition for reasoning about the runtime of quantum programs

• **Flexible:** accommodates multiple runtime models
• **Sensible:** accounts for execution probabilities
• **Expressive:** applies to programs with unbounded loops

Based on existing techniques for probabilistic programs
[Kaminski, Katoen, Matheja & Olmedo - ESOP’16, LICS’16, JACM 65:5]
The programming model (qGCL)
The programming model (qGCL)

Core imperative language over quantum variables with classical control flow
Core imperative language over quantum variables with classical control flow

\[ c ::= q := |b\rangle \quad \text{variable initialization} \]
The programming model (qGCL)

Core imperative language over quantum variables with classical control flow

\[
c ::= \quad q ::= |b\rangle \\
\bar{q} ::= U \bar{q}
\]

variable initialization

unitary transformation
The programming model (qGCL)

Core imperative language over quantum variables with classical control flow

\[
c ::= \quad q := |b\rangle \quad \text{variable initialization}
\]

\[
\overline{q} := U \overline{q} \quad \text{unitary transformation}
\]

\[
\square \; \mathcal{M}[\overline{q}] = m \rightarrow c_m \quad \text{quantum case}
\]
The programming model (qGCL)

Core imperative language over quantum variables with classical control flow

\[ c ::= \quad q ::= |b\rangle \quad \text{variable initialization} \]
\[ \overline{q} ::= U \overline{q} \quad \text{unitary transformation} \]
\[ \Box \mathcal{M}[\overline{q}] = \overline{m \rightarrow c_m} \quad \text{quantum case} \]
\[ \text{while (} \mathcal{M}[\overline{q}] = 1 \text{) do } c \quad \text{quantum loop} \]
The programming model (qGCL)

Core imperative language over quantum variables with classical control flow

\[ c ::= q := |b\rangle \quad \text{variable initialization} \]
\[ \bar{q} := U \bar{q} \quad \text{unitary transformation} \]
\[ \Box M[\bar{q}] = m \rightarrow c_m \quad \text{quantum case} \]
\[ \text{while } (M[\bar{q}] = 1) \text{ do } c \quad \text{quantum loop} \]
\[ \text{skip} \quad \text{no-op} \]
\[ c_1; c_2 \quad \text{sequential composition} \]
Our approach to the runtime of quantum programs
Our approach to the runtime of quantum programs

Goal:

\[ [c] \mathbb{Z} : S \rightarrow \mathbb{R}_\geq 0 \]

\[ [c] \mathbb{Z} \rho = \text{runtime of } c \text{ from initial state } \rho \]
Our approach to the runtime of quantum programs

Goal:

\[
\mathbb{G}_c : S \rightarrow \mathbb{R}_\geq 0
\]

\[
\mathbb{G}_c \rho = \text{runtime of } c \text{ from initial state } \rho
\]

Approach:

Continuation passing style through runtime transformer

\[
\text{ert}[c] : (S \rightarrow \mathbb{R}_\geq 0) \rightarrow (S \rightarrow \mathbb{R}_\geq 0)
\]
Our approach to the runtime of quantum programs

**Goal:**

\[
\lbrack c \rbrack^\mathbb{Z} : S \rightarrow \mathbb{R}_\geq 0
\]

\[
\lbrack c \rbrack^\mathbb{Z} \rho = \text{runtime of } c \text{ from initial state } \rho
\]

**Approach:** Continuation passing style through runtime transformer

\[
\text{ert} [c] : (S \rightarrow \mathbb{R}_\geq 0) \rightarrow (S \rightarrow \mathbb{R}_\geq 0)
\]

runtime of the program following \(c\)
Our approach to the runtime of quantum programs

**Goal:**

\[ [c] \bar{\varepsilon} : S \rightarrow \mathbb{R}_{\geq 0} \]

\[ [c] \bar{\varepsilon} \rho = \text{runtime of } c \text{ from initial state } \rho \]

**Approach:**

Continuation passing style through **runtime transformer**

\[ \text{ert } [c] : (S \rightarrow \mathbb{R}_{\geq 0}) \rightarrow (S \rightarrow \mathbb{R}_{\geq 0}) \]

- runtime of the program following c
- runtime of c, plus the program following c
Our approach to the runtime of quantum programs

**Goal:**

\[
[c] : S \rightarrow \mathbb{R}_{\geq 0}^\infty
\]

\[
[c] \rho = \text{runtime of } c\text{ from initial state } \rho
\]

**Approach:** Continuation passing style through runtime transformer

\[
\text{ert } [c] : (S \rightarrow \mathbb{R}_{\geq 0}^\infty) \rightarrow (S \rightarrow \mathbb{R}_{\geq 0}^\infty)
\]

- runtime of the program following \( c \)
- runtime of \( c \), plus the program following following \( c \)

In particular,

\[
\text{ert } [c] (\lambda \rho'. 0) = [c] \rho
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c.
Transformer \( e_{\text{rt}}[c] \) admits an elegant **definition by induction** on the structure of \( c \)

\[
\begin{align*}
\text{ert}[\overline{q} := U \overline{q}](t) &= \\
\text{ert}[\Box M[\overline{q}] = \overline{m} \rightarrow \overline{c_m}](t) &= \\
\text{ert}[c_1; c_2](t) &=
\end{align*}
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[ \bar{q} := U \bar{q} ; \quad \cdots \cdots \]

\[ e_{rt}[q := U \bar{q}](t) = \]
\[ e_{rt}[\Box \mathcal{M}[\bar{q}] = \overrightarrow{m \rightarrow c_{m}}](t) = \]
\[ e_{rt}[c_{1} ; c_{2}](t) = \]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[
\overline{q} := U \overline{q} ; \quad \cdots \cdots \quad t
\]

\[
\text{ert}[\overline{q} := U \overline{q}](t) = \text{ert}[\square \mathcal{M}[\overline{q} = m \overrightarrow{c_m}](t) = \text{ert}[c_1; c_2](t) =
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[
\bar{q} := U \bar{q} \; ; \; \cdots \cdots \\
\]

\[
\text{ert}[\bar{q} := U \bar{q}](t) = \\
\text{ert}[\square \; \mathcal{M}[\bar{q}] = \overrightarrow{m \rightarrow c_m}](t) = \\
\text{ert}[c_1; c_2](t) = 
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[
\overline{q} := U \overline{q} ; \quad \cdots \cdots \\
\overline{q} \xrightarrow{t} \overline{q} \\
\overline{q} \xrightarrow{T[U]} + t
\]

\[
\text{ert}[\overline{q} := U \overline{q}](t) = \\
\text{ert}[\square \mathcal{M}[\overline{q}] = \overline{m} \xrightarrow{m} \overline{c_m}](t) = \\
\text{ert}[c_1 ; c_2](t) =
\]
Transformer \( \text{ert}[c] \) admits an elegant **definition by induction** on the structure of \( c \)

\[
\bar{q} := U \bar{q} ; \quad \cdots \cdots \quad t \\
T[U] + t \circ [U]
\]

\[
\text{ert}[\bar{q} := U \bar{q}](t) = \\
\text{ert}[\Box \ M[\bar{q}] = m \rightarrow c_m](t) = \\
\text{ert}[c_1; c_2](t) =
\]
Definition of runtime transformer $ert$

Transformer $ert[c]$ admits an elegant **definition by induction** on the structure of $c$

\[
\overline{q} := U \overline{q} ; \quad \cdots \cdots \\
\Rightarrow T[U] + t \circ [U] \\
\lambda \rho. \ U \rho U^\dagger
\]

\[
ert[\overline{q} := U \overline{q}](t) = \\
ert[\square] M[\overline{q} = m \rightarrow c_m](t) = \\
ert[c_1; c_2](t) =
\]
Definition of runtime transformer ert

Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[
\overline{q} := U \overline{q} ; \quad \ldots \ldots
\]

[t]

\[
\mathcal{T}[U] + t \circ [U]
\]

\[
\lambda \rho. \ U \rho U^\dagger
\]

\[
\text{ert}[\overline{q} := U \overline{q}](t)
\]  =  \[
\mathcal{T}[U] + t \circ [U]
\]

\[
\text{ert}[\square \mathcal{M}[\overline{q}] = \overrightarrow{m \rightarrow c_m}](t)
\]  =  \[
\text{ert}[c_1; c_2](t)
\]  =
Definition of runtime transformer \( \text{ert} \)

Transformer \( \text{ert}[c] \) admits an elegant **definition by induction** on the structure of \( c \)

\[
\text{ert}[q := U \overline{q}](t) = T[U] + t \circ [U] \\
\text{ert}[\square \ M[\overline{q}] = m \rightarrow cm](t) = \\
\text{ert}[c_1; c_2](t) =
\]
Transformer \( e_{\text{rt}}[c] \) admits an elegant \textbf{definition by induction} on the structure of \( c \):

\[
\square \mathcal{M}[\overline{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \quad \ldots \ldots
\]

\[
e_{\text{rt}}[\overline{q} := U \overline{q}](t) = T[U] + t \circ [U]
\]

\[
e_{\text{rt}}[\square \mathcal{M}[\overline{q}] = \overline{m \rightarrow c_m}](t) =
\]

\[
e_{\text{rt}}[c_1; c_2](t) =
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[ \Box \mathcal{M}[\overline{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \ldots \]

\[ ert[\Box \mathcal{M}[\overline{q}] = \overline{m} \rightarrow \overline{c_m}](t) = \]

\[ ert[c_1; c_2](t) = \]
Transformer \( \text{ert}[c] \) admits an elegant **definition by induction** on the structure of \( c \)

\[
\square \mathcal{M}[\overline{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \quad \ldots \ldots
\]

\[
\mathcal{T}[\mathcal{M}]
\]

\[
\text{ert}[\overline{q} := U \overline{q}](t) = \mathcal{T}[U] + t \circ [U]
\]

\[
\text{ert}[\square \mathcal{M}[\overline{q}] = m \rightarrow c_m](t)
\]

\[
\text{ert}[c_1 ; c_2](t)
\]
Transformer \( \text{ert}[c] \) admits an elegant **definition by induction** on the structure of \( c \)

\[
\begin{align*}
\square M[\bar{q}] &= m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \quad \ldots \ldots \\
T[M] + \text{ert}[c_1](t) + \text{ert}[c_2](t)
\end{align*}
\]

\[
er[t]_{\bar{q} := U \bar{q}}(t) = T[U] + t \circ \llbracket U \rrbracket
\]

\[
er[\square M[\bar{q}] = m \rightarrow c_m](t) = \text{ert}[c_1; c_2](t) =
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[ \square \mathcal{M}[\overline{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \ldots \ldots \]

\[ \mathcal{T}[\mathcal{M}] + \text{ert}[c_1](t) \circ [\mathcal{M}=m_1] + \text{ert}[c_2](t) \circ [\mathcal{M}=m_2] \]

\[ \text{ert}[\overline{q} := U \overline{q}](t) = \mathcal{T}[U] + t \circ [U] \]
\[ \text{ert}[\square \mathcal{M}[\overline{q}] = \overline{m} \rightarrow c_m](t) = \]
\[ \text{ert}[c_1; c_2](t) = \]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[ \square \mathcal{M}[\bar{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \ldots \ldots \]

\[ t \]

\[ T[\mathcal{M}] + \Pr[\mathcal{M}=m_1] \cdot ert[c_1](t) \circ [\mathcal{M}=m_1] + \Pr[\mathcal{M}=m_2] \cdot ert[c_2](t) \circ [\mathcal{M}=m_2] \]

\[ ert[\bar{q} := U \bar{q}](t) = T[U] + t \circ [U] \]

\[ ert[\square \mathcal{M}[\bar{q}] = \overrightarrow{m \rightarrow c_m}](t) = \]

\[ ert[c_1; c_2](t) = \]
Definition of runtime transformer ert

Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[
\begin{align*}
\square \, \mathcal{M}[\bar{q}] &= m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 \quad ; \quad \ldots \ldots \\

\mathcal{T}[\mathcal{M}] + \Pr[\mathcal{M}=m_1] \cdot \text{ert}[c_1](t) \circ [\mathcal{M}=m_1] + \Pr[\mathcal{M}=m_2] \cdot \text{ert}[c_2](t) \circ [\mathcal{M}=m_2]
\end{align*}
\]

\[
\begin{align*}
\text{ert}[\bar{q} := U \, \bar{q}](t) &= \mathcal{T}[U] + t \circ [U] \\
\text{ert}[\square \, \mathcal{M}[\bar{q}] = m \rightarrow c_m](t) &= \mathcal{T}[\mathcal{M}] + \sum_m \Pr[\mathcal{M}=m] \cdot \text{ert}[c_m](t) \circ [\mathcal{M}[\bar{q}]] \\
\text{ert}[c_1; c_2](t) &= \quad =
\end{align*}
\]
Definition of runtime transformer ert

Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[ \square \mathcal{M}[\overline{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \ldots \ldots \]

\[ \mathcal{T}[\mathcal{M}] + \Pr[\mathcal{M}=m_1] \cdot ert[c_1](t) \circ [\mathcal{M}=m_1] + \Pr[\mathcal{M}=m_2] \cdot ert[c_2](t) \circ [\mathcal{M}=m_2] \]

\[
\begin{align*}
ert[\overline{q} := U \overline{q}](t) & = \mathcal{T}[U] + t \circ [U] \\
ert[\square \mathcal{M}[\overline{q}] = \overline{m} \rightarrow \overline{c_m}](t) & = \mathcal{T}[\mathcal{M}] + \sum_m \Pr[\mathcal{M}=m] \cdot ert[c_m](t) \circ [\mathcal{M}[\overline{q}]] \\
ert[c_1; c_2](t) & = ert[c_1](ert[c_2](t))
\end{align*}
\]
Transformer ert[c] admits an elegant **definition by induction** on the structure of c

\[ \square \mathcal{M}[\overline{q}] = m_1 \rightarrow c_1 \mid m_2 \rightarrow c_2 ; \ldots \ldots \]

\[ T[\mathcal{M}] + \operatorname{Pr}[\mathcal{M}=m_1] \cdot \operatorname{ert}[c_1](t) \circ [\mathcal{M}=m_1] + \operatorname{Pr}[\mathcal{M}=m_2] \cdot \operatorname{ert}[c_2](t) \circ [\mathcal{M}=m_2] \]

\[ \operatorname{ert}[\overline{q} := U \overline{q}](t) = T[U] + t \circ [U] \]

\[ \operatorname{ert}[\square \mathcal{M}[\overline{q}] = \overline{m} \rightarrow \overline{c_m}](t) = T[\mathcal{M}] + \sum_m \operatorname{Pr}[\mathcal{M}=m] \cdot \operatorname{ert}[c_m](t) \circ [\mathcal{M}[\overline{q}]] \]

\[ \operatorname{ert}[c_1; c_2](t) = \operatorname{ert}[c_1](\operatorname{ert}[c_2](t)) \]

\[ \vdots \]
We can establish upper bounds for the runtime of loops using a notion of loop invariant:
Invariant-based reasoning for the runtime of loops

We can establish upper bounds for the runtime of loops using a notion of loop invariant:

\[
F_t^{\langle \mathcal{M},c \rangle} (I) \leq I
\]

\[
\text{ert[while } (\mathcal{M} = 1) \text{ do } c](t) \leq I
\]

\rightarrow I \text{ is a loop invariant}

\rightarrow I \text{ is an upper bound of the loop runtime}
Case study: BB84 quantum key distribution algorithm
Case study: BB84 quantum key distribution algorithm

**Goal:** securely create and distribute a shared (symmetric) key between two parties.
**Case study: BB84 quantum key distribution algorithm**

**Goal:** securely create and distribute a shared (symmetric) key between two parties.

```
\\ initialize counter
k := \langle 0 \rangle;
\\ while not reached m bits
while (M[k] = 1) do
  \\\
  \\\ measure Alice’s coins
  A := \langle ++ \rangle; B := \langle + \rangle;
  \\\ measure Alice’s coins
  \square \cdot M_A[A] =
  \\\ measure Bob’s coin
  \langle eb \rangle \rightarrow \square \cdot M_B[B]
  \\\ if Alice’s and Bob’s basis agree
  \\\ store bit b and increment counter
  \langle e \rangle \rightarrow k, Q := U_{pb}[k, Q];
  k := U_{>}[k];
  \\\ if Alice’s and Bob’s basis disagree
  \\\ discard bit b
  \langle \neg e \rangle \rightarrow skip
```
Case study: BB84 quantum key distribution algorithm

**Goal:** securely create and distribute a shared (symmetric) key between two parties.

```
// initialize counter
k := |0⟩;

// while not reached m bits
while (M[k] = 1) do
    // flip Alice’s and Bob’s coins
    A := |++⟩; B := |+⟩;
    // measure Alice’s coins
    □ · M_A[A] =
        // measure Bob’s coin
        |eb⟩ → □ · M_B[B]
            // if Alice’s and Bob’s basis agree
            // store bit b and increment counter
        |e⟩ → k. Q := U_p[k, Q];
            // k := U_p[k];
        |¬e⟩ → skip
```

**Average time required to generate a key of m bits:**

\[
\mathcal{T}[|0⟩] + 2\mathcal{T} m + \mathcal{T}[M] \in \mathcal{O}(m)
\]
Case study: BB84 quantum key distribution algorithm

**Goal:** securely create and distribute a shared (symmetric) key between two parties.

\[
\begin{align*}
\text{\texttt{initialize counter}} & \quad k := \lvert 0 \rangle; \\
\text{\texttt{while not reached m bits}} & \\
\text{\texttt{while}} & \quad (M[k] = 1) \text{ do} \\
\text{\texttt{flip Alice’s and Bob’s coins}} & \quad A := \lvert ++ \rangle; \quad B := \lvert + \rangle; \\
\text{\texttt{measure Alice’s coins}} & \quad \Box \cdot M_A[A] = \\
\text{\texttt{measure Bob’s coin}} & \quad \lvert eb \rangle \rightarrow \Box \cdot M_B[B] \\
\text{\texttt{if Alice’s and Bob’s basis agree}} & \quad \text{\texttt{store bit b and increment counter}} \\
\text{\texttt{if}} & \quad \text{\texttt{if Alice’s and Bob’s basis disagree}} \\
\text{\texttt{if}} & \quad \text{\texttt{discard bit b}} \\
\text{\texttt{if}} & \quad \text{\texttt{skip}}
\end{align*}
\]

**Average time required to generate a key of m bits:**

\[
\mathcal{T}[\lvert 0 \rangle] + 2\mathcal{T} m + \mathcal{T}[M] \in O(m)
\]

counter \( k \) initialization
**Case study: BB84 quantum key distribution algorithm**

**Goal:** securely create and distribute a shared (symmetric) key between two parties.

Average time required to generate a key of $m$ bits:

$$\mathcal{T}[|0\rangle] + 2\mathcal{T} m + \mathcal{T}[\mathcal{M}] \in \mathcal{O}(m)$$

**Algorithm**

- **Initialization**
  
  ```python
  # initialize counter
  k := |0\rangle;
  # while not reached m bits
  while (M[k] = 1) do
    # flip Alice's and Bob's coins
    A := |++\rangle; B := |+\rangle;
    # measure Alice's coins
    □ · M_A[A] =
      # measure Bob's coin
      |eb⟩ → □ · M_B[B]
      # if Alice's and Bob's basis agree
      store bit b and increment counter
      |e⟩ → k, Q := U_{P_b}[k, Q];
      k := U_{>}[k];
      # if Alice's and Bob's basis disagree
      discard bit b
      |¬e⟩ → skip
  ```
**Case study: BB84 quantum key distribution algorithm**

**Goal:** securely create and distribute a shared (symmetric) key between two parties.

```
\| initialize counter
k := |0⟩;
\| while not reached m bits
while (M[k] = 1) do
  \| flip Alice’s and Bob’s coins
  A := |++⟩; B := |+⟩;
  \| measure Alice’s coins
  □ \cdot M_A[A] =
    \| measure Bob’s coin
    |eb⟩ → □ \cdot M_B[B]
      \| if Alice’s and Bob’s basis agree
      \| store bit b and increment counter
      |e⟩ → k, Q := U_{P_b}[k, Q];
        k := U_{ϕ}[k];
      \| if Alice’s and Bob’s basis disagree
      \| discard bit b
      |¬e⟩ → skip
```

Average time required to generate a key of $m$ bits:

$$T[|0⟩] + 2T m + T[M] \in \mathcal{O}(m)$$
**Case study: BB84 quantum key distribution algorithm**

**Goal:** securely create and distribute a shared (symmetric) key between two parties.

\[\begin{align*}
\text{initialize counter} \\
k & := |0\rangle; \\
\text{while not reached } m \text{ bits} \\
\text{while } (M[k] = 1) \text{ do} \\
\text{flip Alice’s and Bob’s coins} \\
A & := |++\rangle; \\
B & := |+\rangle; \\
\text{measure Alice’s coins} \\
\text{measure Bob’s coin} \\
|e\rangle & \rightarrow \square \cdot M_B[B] \\
\text{if Alice’s and Bob’s basis agree} \\
\text{store bit } b \text{ and increment counter} \\
|e\rangle & \rightarrow k, Q := U_{pb}[k, Q]; \\
\text{increment counter} \\
k & := U_{\gg}[k]; \\
\text{if Alice’s and Bob’s basis disagree} \\
\text{discard bit } b \\
|\neg e\rangle & \rightarrow \text{skip}
\end{align*}\]

**Average time required to generate a key of** \(m\) **bits:**

\[T[|0\rangle] + 2T m + T[M] \in \mathcal{O}(m)\]

\[T = T[M] + T[|++\rangle] + T[|+\rangle] + T[M_A] + T[M_B] + \frac{1}{2}(\frac{1}{2}T[U_0] + \frac{1}{2}T[U_1] + T[U_{\gg}]) + \frac{1}{2}\]
Conclusion
First step to a *formal and compelling* resource analysis of quantum programs

Existing techniques for probabilistic programs extend smoothly to quantum programs.
First step to a *formal and compelling* resource analysis of quantum programs

Existing techniques for probabilistic programs extend smoothly to quantum programs

**Future work**

- Connection to an operational model
- Language extensions
- Automation
First step to a *formal and compelling* resource analysis of quantum programs

Existing techniques for probabilistic programs extend smoothly to quantum programs

**Future work**

- Connection to an operational model
- Language extensions
- Automation

**Thanks!**
BACKUP SLIDES
Language semantics

\[
\begin{align*}
\llbracket \text{skip} \rrbracket(\rho) &= \rho \\
\llbracket q := |b\rangle \rrbracket(\rho) &= \rho[q \mapsto |b\rangle] \\
\llbracket \overline{q} := U \overline{q} \rrbracket(\rho) &= U^{\dagger}q \rho U^{\dagger}q^{\dagger} \\
\llbracket c_1; c_2 \rrbracket(\rho) &= \llbracket c_1 \rrbracket(\llbracket c_2 \rrbracket(\rho)) \\
\llbracket \square \ M[\overline{q}] = \overline{m} \rightarrow c_m \rrbracket(\rho) &= \sum_m \Pr_{\rho}[M=m] \cdot \llbracket c_m \rrbracket(\rho|_{M=m}) \\
\llbracket \text{while } (M[\overline{q}] = 1) \text{ do } c \rrbracket(\rho) &= \text{lfp}\left(\Phi^{M,c}\right)
\end{align*}
\]

\[
\lambda X. \lambda \rho'. \Pr_{\rho'}[M=0] \cdot \rho'|_{M=0} + \sum_{M=1} \Pr_{\rho'}[M=1] \cdot \llbracket c \rrbracket(\rho'|_{M=1})
\]

\[
\frac{M_m \rho M_m^{\dagger}}{\text{tr}(M_m^{\dagger} M_m \rho)}
\]

\[
\frac{\text{tr}(M_m^{\dagger} M_m \rho)}{M_m \rho M_m^{\dagger}}
\]
The results follow from a direct application of Park's Theorem \cite{12}, exploiting the runtime of programs (in particular, of quantum programs). In the second rule, the resulting state is that transforms the dimension of the density matrix in the runtime of programs (in particular, of quantum programs). Here, runtime transformer \( M \) takes as input a state \( \rho \) and its type \( M \), where \( \rho \) is an unitary operator denoted \( U \). The resulting state is computed as the least fixed point of transformers, which is not a simple task. Nevertheless, to recover the expected runtime of programs, we use a continuation-based approach to model the runtime of programs. Runtime transformer \( M \) is a variable initialization with vector \( \text{lfp}(F_t^{M[q]}, c) \)

\[
\text{ert}[\text{skip}](t) = \lambda \rho. 1 + t(\rho)
\]

\[
\text{ert}[q := |b\rangle](t) = \lambda \rho. T[|b\rangle] + t(\rho[q \mapsto |b\rangle])
\]

\[
\text{ert}[\bar{q} := U \bar{q}](t) = \lambda \rho. T[U] + t(U^\dagger \rho U^\dagger \dagger)
\]

\[
\text{ert}[c_1; c_2](t) = \text{ert}[c_1](\text{ert}[c_2](t))
\]

\[
\text{ert}[\square M[\bar{q}] = \overline{m} \rightarrow c_m](t) = \lambda \rho. T[M] + \sum_m \text{Pr}_\rho[M=m] \cdot \text{ert}[c_m](t)(\rho|_{M=m})
\]

\[
\text{ert[while } (M[\bar{q}] = 1) \text{ do } c](t) = \text{lfp}(F_t^{M[q]}, c)
\]

\[
\lambda t'. \lambda \rho. T[M] + \text{Pr}_\rho[M=1] \cdot \text{ert}[c](t')(\rho|_{M=1}) + \text{Pr}_\rho[M=0] \cdot t(\rho|_{M=0})
\]