A high-level quantum programming language with dynamical types to represent Hilbert space bases (defense)

Mads J. Damgaard

June 19, 2020

Goal of the programming language

- To make an abstraction so that quantum data and classical data are treated equally.
- This is beneficial since quantum algorithms often contain classical subroutines.
- It also means that the compiler gets to decide whether to run subroutines on classical or quantum hardware.

My solution

- A language that uses dynamic types to represent the bases of the quantum states.
- It uses Haskell-like type signatures with syntax to show how the types change.

```
foo :: bool -> ()bool* -> (xspin*)bool* -> bool
```

My solution

- A language that uses dynamic types to represent the bases of the quantum states.
- It uses Haskell-like type signatures with syntax to show how the types change.
- Has a procedural inner syntax inspired by C and Quipper.

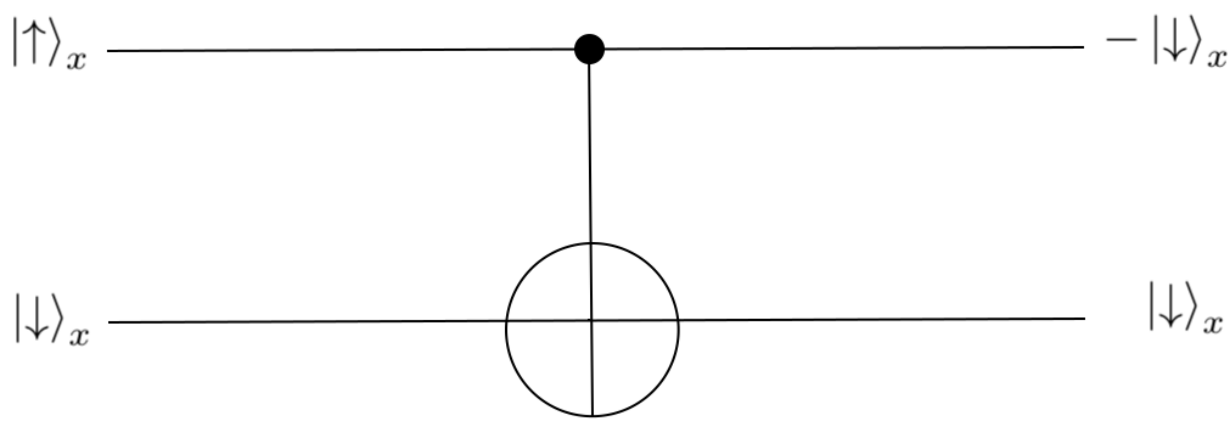
```
foo :: bool -> ()bool* -> (xspin*)bool* -> bool
foo :: a b c {
    bool x;
    x <- b;
    if (a) {
        Not x;
    }
    Had c;
    return x;
}
```

The if statement

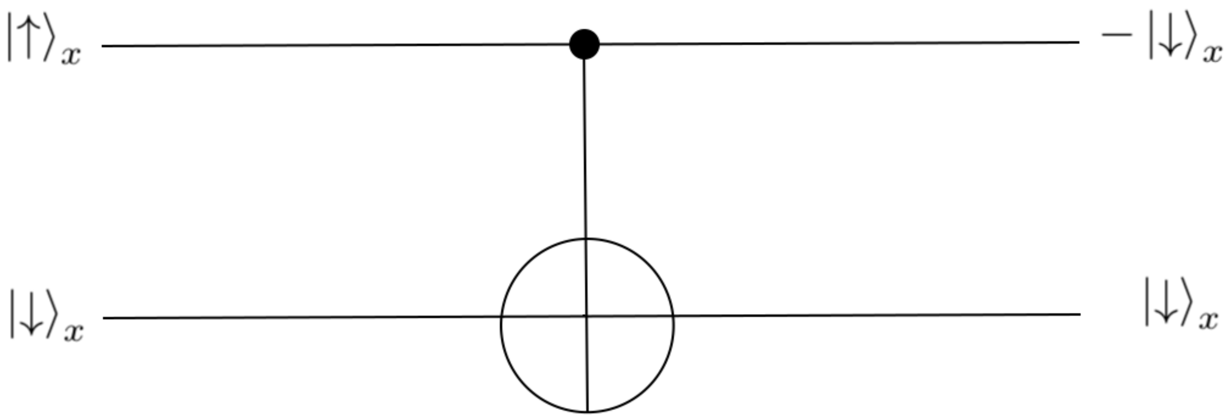
- It gives a classical abstraction over controlled circuits.
- It uses the special Boolean types for condition variables.
- It does not change the condition variables iff they are Boolean.

```
Tof :: bool -> bool -> bool* -> ()
Tof :: a b c {
    if (a && b) {
        Not c;
    }
}
```

A controlled circuit (CNOT) in the x-basis



A controlled circiut (CNOT) in the x-basis



$$|\uparrow\rangle_x \otimes |\downarrow\rangle_x = \begin{pmatrix}1\\1\end{pmatrix} \otimes \begin{pmatrix}1\\-1\end{pmatrix} = \begin{pmatrix}1\\-1\\1\\-1\end{pmatrix} : \begin{matrix}|\uparrow\uparrow\rangle_z\\|\uparrow\downarrow\rangle_z\\|\downarrow\uparrow\rangle_z\\|\downarrow\downarrow\rangle_z\end{matrix}$$

$$\begin{pmatrix}1\\-1\\1\\-1\end{pmatrix} \rightarrow \begin{pmatrix}0&1&0&0\\1&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}1\\-1\\1\\-1\end{pmatrix} = \begin{pmatrix}-1\\1\\1\\-1\end{pmatrix} : \begin{matrix}|\uparrow\uparrow\rangle_z\\|\uparrow\downarrow\rangle_z\\|\downarrow\uparrow\rangle_z\\|\downarrow\downarrow\rangle_z\end{matrix}$$

$$\begin{matrix}|\uparrow\uparrow\rangle_z\\|\uparrow\downarrow\rangle_z\\|\downarrow\uparrow\rangle_z\\|\downarrow\downarrow\rangle_z\end{matrix} : \begin{pmatrix}-1\\1\\1\\-1\end{pmatrix} = -\begin{pmatrix}1\\1\end{pmatrix} \otimes \begin{pmatrix}1\\-1\end{pmatrix} = -\,|\downarrow\rangle_x \otimes |\downarrow\rangle_x$$

The temp and restore statements

- Ancilla qubits are important for quantum algorithms.
- These need to be uncomputed correctly.
- This language solution takes care of this safely with the temp and restore statements.
- They can be used to define local variables and make temporary changes to existing ones.
- Local or changed variables <u>cannot</u> be changed afterwards, except temporarily.

```
RippleCarryAdder :: bool[4] -> bool[4]* -> bool* -> ()
RippleCarryAdder a[0:3] b[0:3] z {
    // (s[0], s[1], s[2], s[3]) will end up as the sum (mod 16).
    b[0:3] -> s[0:3];
    // z' will end up as z ^ c_out.
    z -> z';
    // compute each carry-out progressively and then compute the
    // sum in reverse while restoring each necessary carry-outs on
    // the way.
    bool[4] c[0:3];
    temp c[0] <- new false;
        temp c[1] <- majority a[0] b[0] c[0];
            temp c[2] <- majority a[1], b[1], c[1];
                temp c[3] <- majority a[2], b[2], c[2];
                    temp c[4] <- majority a[3], b[3], c[3];
                        z' <- CNOT c[4] . z;
                    restore;
                    s[3] <- CNOT c[3] . CNOT a[3] . b[3];
                restore;
                s[2] <- CNOT c[2] . CNOT a[2] . b[3];
            restore;
            s[1] <- CNOT c[1] . CNOT a[1] . b[1];
        restore;
        s[0] <- CNOT c[0] . CNOT a[0] . b[0];
    restore;
}
```

Garbage and measurements

- Garbage can be created when some output is not needed.
- It can be restored if it is generated in an temp statement.
- Measurements cannot be restored.

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    bool g1 = CNOT a . b;
    bool g2 = CNOT a . c_in;
    if (g1 && g2) {
        Not a;
    }
    return a;
}
```

```
isBalanced<n> :: (bool[n] -> bool) --> bool + garbage
isBalanced<n> f {
    xspin[n] x[1:n] = Hadamards<n> . new 0;
    xspin y = Had . new 1;
    //
    temp bool a = f x[1:n];
    if (a) {
        Not y;
    }
    restore;
    // assert that y will that y will have qbit type by now.
    assert y of qbit;
    // make another Hadamard tranformations of the xs.
    Hadamards<n> x[1:n];
    // measure y to see if the output is 0 or not.
    let m = measure x[1:n] in {
        if (m == 0) {
            return new false;
        } else {
            return new true;
        }
    }
}
```

Aliases

- Aliases can be used to change variable names despite the often limited amount of variables.
- It makes it possible to still disallow for variables to leave their local scope.

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    bool[2] (g1, g2);
    a -> maj;
    (g1, g2, maj) <- Tof . (CNOT a . b, CNOT a . c_in, a)
    return maj;
}
```

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    // declare aliases xor_ab for b and xor_ac for c.
    b    -> xor_ab;
    c_in -> xor_ac;
    // compute xors.
    if (a) {
        xor_ab <- Not . b;
        xor_ac <- Not . c_in;
    }
    // declare alias maj for a.
    a -> maj;
    // compute majority.
    if (xor_ab && xor_ac) {
        maj <- Not . a;
    }
    // discard xor_ab and xor_ac (i.e. b and c_in).
    discard xor_ab;
    discard xor_ac;
    // return the majority (consuming the input parameter).
    return maj;
}
```

Aliases

- Aliases can be used to change variable names despite the often limited amount of variables.
- It makes it possible to still disallow for variables to leave their local scope.

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    bool[2] (g1, g2);
    a -> maj;
    (g1, g2, maj) <- Tof . (CNOT a . b, CNOT a . c_in, a)
    return maj;
}
```

```
RippleCarryAdder :: bool[4] -> bool[4]* -> bool* -> ()
RippleCarryAdder a[0:3] b[0:3] z {
    // (s[0], s[1], s[2], s[3]) will end up as the sum (mod 16).
    b[0:3] -> s[0:3];
    // z' will end up as z ^ c_out.
    z -> z';
    // compute each carry-out progressively and then compute the
    // sum in reverse while restoring each necessary carry-outs on
    // the way.
    bool[4] c[0:3];
    temp c[0] <- new false;
        temp c[1] <- majority a[0] b[0] c[0];
            temp c[2] <- majority a[1], b[1], c[1];
                temp c[3] <- majority a[2], b[2], c[2];
                    temp c[4] <- majority a[3], b[3], c[3];
                        z' <- CNOT c[4] . z;
                    restore;
                    s[3] <- CNOT c[3] . CNOT a[3] . b[3];
                restore;
                s[2] <- CNOT c[2] . CNOT a[2] . b[3];
            restore;
            s[1] <- CNOT c[1] . CNOT a[1] . b[1];
        restore;
        s[0] <- CNOT c[0] . CNOT a[0] . b[0];
    restore;
}
```

Type checking rules

- I have used as a kind of pseudocode with similarities to the logic programming paradigm.

$$\frac{\begin{array}{c}\vdash \gamma_0 = \gamma \quad \vdash \tau_0 = \tau \\ \forall_{i\in\{1,\ldots,n\}}\left[\tau_{i-1}, \gamma_{i-1} \vdash e_i : t_i \hookrightarrow \Delta\tau_i\right] \\ \forall_{i\in\{1,\ldots,n\}}\left[\tau_{i-1} \vdash \gamma_i = \lfloor e_i \rfloor \diamond \gamma_{i-1} \ \wedge \ \tau_i = [\lfloor e_i \rfloor : \mathbf{c}] \circ \Delta\tau_i \circ \tau_{i-1}\right] \\ \vdash \Delta\tau = \Delta\tau_n \circ \cdots \circ \Delta\tau_1 \circ [\,]\end{array}}{\tau, \gamma \vdash (e_n, \ldots, e_1) : (t_n, \ldots, t_1) \hookrightarrow \Delta\tau}$$

$$\frac{\begin{array}{c}\vdash s = \texttt{if}\ (c)\ s' \\ \tau \vdash c \hookrightarrow \Delta\tau_{in} \\ \Delta\tau_{in} \circ \tau \vdash s' : \text{none} \hookrightarrow \Delta\tau' \\ \vdash \text{unrestored}(\Delta\tau') = 0 \\ \vdash [\text{garbage}] \notin \Delta\tau' \qquad \vdash [\text{measurement}] \notin \Delta\tau' \\ \vdash \tau' = \Delta\tau' \circ \tau \\ \forall_{v\in\|\Delta\tau'\|_{vars}}\left[\tau \vdash v : t_v^{in} \qquad \tau' \vdash v : t_v^{out}\right] \\ \forall_{v\in\|\Delta\tau'\|_{vars}}\left[\tau \vdash \text{span}(t_v^{in}) = \text{span}(t_v^{out})\right] \\ \vdash \{v_1, \ldots, v_n\} = \{v \in \|\Delta\tau'\|_{vars} \mid t_v^{out} \neq t_v^{in}\} \\ \forall_{i\in\{1,\ldots,n\}}\left[\vdash \Delta\tau_i = [v_i : \text{span}(t_{v_i}^{in})]\right] \\ \vdash \Delta\tau = \Delta\tau_n \circ \cdots \circ \Delta\tau_1 \circ \Delta\tau' \circ [\,]\end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}$$

Questions

Extra material

```
bar :: bool -> ()
bar a {
    temp bool aIsZero = isZero a;
    ...
    temp Not aIsZero -> aIsNonZero;
    ...
    restore;
    restore;
}
```

$$\frac{\begin{array}{c}\vdash s = \texttt{temp}\ s'\ ; \\ [\,\|\tau\|_{vars} : *\,] \circ \tau \vdash s' \hookrightarrow \Delta\tau' \\ \vdash \text{unrestored}(\Delta\tau') = 0 \\ \vdash [\text{measurement}] \notin \Delta\tau' \\ \vdash \{v_1, \ldots, v_n\} = \|s'\|_{vars} \\ \vdash \tau' = \Delta\tau' \circ [\,\|\tau\|_{vars} : *\,] \circ \tau \\ \forall_{i \in \{1,\ldots,n\}}\ [\tau' \vdash v_i : t_i] \\ \vdash \Delta\tau = \big[\,[\{v_1, \ldots, v_n\} : \mathbf{c}\,] \circ [v_1 : t_1] \circ \cdots \circ [v_n : t_n]\,\big]_{temp}\end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}$$

$$\texttt{bool[3]} = \texttt{(bool, bool, bool)}$$
$$\texttt{a[2:4]} = \texttt{(a[2], a[3], a[4])}$$