Introduction to Quantum Programming and Semantics

Lecture 17: Uncomputation

Chris Heunen

University of Edinburgh

Overview

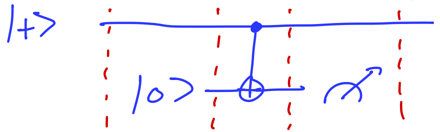
- Uncomputation
- Silq

Uncomputation

Uncomputation

with probability $1/2$: $|0\rangle$

with probability $1/2$: $|1\rangle$

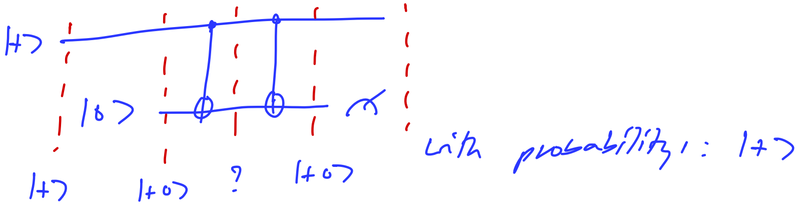


$|+\rangle$

$|+0\rangle = |+\rangle \otimes |0\rangle = \frac{(|1\rangle + |0\rangle)}{\sqrt{2}} \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$

$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$

Uncomputation



Silq

Silq

- Open source

- Similar to Q#
  - can mix classical and quantum computations

- Sophisticated type system can distinguish
  - classical and quantum types
  - classical and quantum subroutines
  - quantum subroutines with and without measurement

- Automatic uncomputation

QRAM model

- Silq programs run on classical computer controlling quantum computer
  - Send quantum instructions
  - Receive measurement results
  - Continue depending on results

Silq type system

- Basic types:
    - B (Booleans)
    - N (Natural numbers)
    - Z (integers)
    - int[n], uint[n] (n-bit signed and unsigned integers)
- Type constructors:
    - s->t (function types)
    - s[] (lists)
    - s^n (tuples)
    - !s (restriction to classical values)

Silq type system

```
def discard (n : !N) {
  return true;
}
```

```
def discardQuantum (n : N) {
  return true;
}
```

measure : t -> !t

(polymorphic in t)

Silq annotation system

- s -> mfree t
  functions that do not measure any (parts of) arguments
  guarantees that superposition is not destroyed

- s -> qfree t
  functions that to not introduce or eliminate superpositions
  guarantees that superposition not changed
  very useful for classical oracles

Silq generic parameters

- Generic parameter = classical value known at compile-time
- Functions may depend on generic parameters

```
def bitwise_not [n:!N] (bits : B^n) qfree {
  for i in [0..n) {
    bits[i] := X(bits[i]);
  }
  return bits;
}

def main() {
  xs := bitwise_not(false, false, true);
  ys := bitwise_not(true, true);
  return (xs,ys);
}
```

Silq generic parameters

- Generic parameter = classical value known at compile-time
- Functions may depend on generic parameters

```
def bitwise_map [n:!N] (bits : B^n, f : !(B -> B)) {
  for i in [0..n) {
    bits[i] := f(bits[i]);
  }
  return bits;
}

def main() {
  xs := bitwise_map((false, true), H);
  return xs;
}
```

Toy example

```
def discard (n : !N) {
    return true;
}

def discardQuantum (n : N) {
    return true;
}

def bitwise_not [n:!N] (bits : B^n) qfree {
    for i in [0..n) {
        bits[i] := X(bits[i]);
    }
    return bits;
}

def bitwise_map [n:!N] (bits : B^n, f : !(B->B)) {
    for i in [0..n) {
        bits[i] := f(bits[i]);
    }
    return bits;
}

def main () {
    zs := bitwise_map((false,true),H);
    return zs;
    // xs := bitwise_not[3](false,true,false);
    // ys := bitwise_not(true,true);
    // return (xs,ys);
}
```

Deutsch-Josza

```
def DeutschJozsa[n:!N](f : B^n !-> lifted !B) {
    x:=0:int[n];
    for i in [0..n) { x[i] := H(x[i]); }
    if f(x as B^n) { phase(pi); }
    for i in [0..n) { x[i] := H(x[i]); }
    return measure(x)==0;
}

def oracle(xs : B^8) lifted {
    return true;
}

def main() {
    return DeutschJozsa(oracle);
}
```

Summary:

- Uncomputation necessary to (re)use auxiliary qubits
- Can be done automatically
- Needs annotations to help compiler
- Silq does this in a conservative way