Quantum Superpositions and the First Virtue

Damian Conway

School of Computer Science and Software Engineering
Monash University
Australia

Review of QM

• The Mars model
• In a quantised system entities can exist only in discrete states
• Values are non-continuous
• Like finite precision arithmetic...
• What number(s) are between 3.1415926 and 3.1415927?
• In 32-bit arithmetic: none!
• Floating point numbers have a non-continuous domain.
• So too the entire universe (if we look closely enough)
• Niels Bohr
• Fundamental building blocks of matter are wave-like phenomena...
• ..but quantised so they’re only observable at discrete mass/energy levels
• Albert Einstein
• The quantised mass/energy levels of wave-like phenomena are observed as particles.
• For example, when you observe an electromagnetic wave, you see a photon.
• Wonderful, wonderful Copenhagen Interpretation
• If it can be in state $\alpha$, $\beta$, or $\gamma$ when someone looks at it...
• …then its in all three states simultaneously until someone does!
Superpositions

- F-1 model
- On the curve, Fangio was simultaneously in the potential states of:
  - “absolutely certain to win” and
  - “skidding wildly out of control in a flaming pile of twisted metal”
- Until someone looked at him
- At which point his delicately balanced multiplicity of states “collapsed” into a single actual state
- This final resting state is known as an eigenstate
- From the German *Eigen*, meaning “proper, actual, real”
- So it is with photons flying through slits: they’re everywhere until you’re actually looking for one;
- So too electrons tunneling across semiconductors: they leak away until you start using them;
- Schrödinger’s poor long-suffering cat...
- Existentially uncertain until young Erwin lifted the lid
- Can represent this condition as a ghostly overlay (a superposition) of states within a single entity...

A superposition isn’t a combination of states
- It’s both states at once (possibly in two separate parallel universes)
Review of QC

- Quantum computing
- Store a bit (1 or 0) in the state of a single particle
- For example, its spin
- Now arrange the particle’s state as a superposition of the “1” state and the “0” state.
- You have a quantum bit (or qubit) storing both 1 and 0 simultaneously
- Repeat with a second particle
- Now arrange for the two particles to interact in a manner equivalent to an AND gate.
- As long as you don’t look at it, the result is another superposition of all the possible outcomes
- But now the superposed “0” state is three times liker than the superposed “1” state
- We can do better than that
- Define appropriate quantum operations that result in undesirable states having zero probability
- Then look at the result and it must collapse to the desired answer
- In 1994, Peter Shor proposed a series of quantum operations for factoring an arbitrarily large number
- Overnight quantum physics went from a refuge for lunatics, to an...ISSUE OF NATIONAL SECURITY!!!
- The problems are...
- Hardware-wise, it’s damn difficult to superimpose quanta
- And even harder to keep them superimposed
- Software-wise, the math makes most hackers teeth ache
- \( U_{SRN} \) is \( \sqrt{\neg} \)
- That is: \( U_{SRN} (|1\rangle) \rightarrow |\text{noise}\rangle \)
  \( U_{SRN} (|\text{noise}\rangle) \rightarrow |0\rangle \)
- And vice versa
- It a nightmare
- So it will be some time yet before you can get your einMac or your Windows QT system
- And it will probably be even longer before Larry approves a \( \text{sqrt!}() \) builtin.
Quantum Scalars

• Suppose it wasn’t the individual bits that were superimposed
• Suppose it was complete scalar values instead
• Then we wouldn’t have to be a mathematician/physicist/chemist/engineer to set up a quantum computation
• We could just add a few superimposing operators to Perl…
• And start hacking in multiple parallel universes!
• Two new operators: any and all
• Each takes a list of scalar values…
• …and superimposes them to create a single scalar value
• A superposition
• Two distinct kinds of superposition
• any creates a disjunctive superposition
• Acts as if it is in any of its superimposed states, according to need
• all creates a conjunctive superposition
• Acts as if it is in all of its superimposed states simultaneously

Quantum Arithmetic

• Superpositions can participate in arithmetic and logical operations just like any other Perl scalar
• Operations are applied to superpositions in parallel…
• …to each of the states in that superposition.
• For example:

```perl
use Quantum::Superpositions;

my $result = any(1,2,3) * 2;
```

• Result is a superposition of states 2, 4, and 6.
• Which we could then use in other computations:

```perl
if ($result == 4) {
    print "fore!"
}
```

• That comparison also returns a superposition
• One that is both true and false

• Need some mechanism to resolve which superimposed state the if responds to

• Solution is the two types of superposition available

• Disjunctive superposition is true if any of its states is true

• Conjunctive superposition is true only if all of its states are true.

• Thus "fore!" is printed

```perl
    if ($result == 4) {
        print "fore!"
    }
```

• Equivalent to:

```perl
    if (any(1,2,3) * 2 == 4) {
        print "fore!"
    }
```

• Equivalent to:

```perl
    if (any(2,4,6) == 4) {
        print "fore!"
    }
```

• Equivalent to:

```perl
    if (any(0,1,0)) {
        print "fore!"
    }
```

• Thus "fore!" is printed

```perl
    if ($result == 4) {
        print "fore!"
    }
```

• On the other hand:

```perl
    if (all(1,2,3) * 2 == 4)...
```

• Equivalent to:

```perl
    if (all(2,4,6) == 4)...
```

• Equivalent to:

```perl
    if (all(0,1,0))...
```

• Fails
• Operations also possible between two superpositions:

```perl
print "no alcohol\n"
  if all(1,2,3)*any(5,6) < 21;

print "no entry\n"
  if all(1,2,3)*any(5,6) < 18;

print "underage\n"
  if any(1,2,3)*all(5,6) < 18;
```

• Equivalent to:

```perl
print "no alcohol\n"
  if all(5,6,10,12,15,18) < 21;

print "no entry\n"
  if all(5,6,10,12,15,18) < 18;

print "underage\n"
  if any(5,6,10,12,15,18) < 18;
```

• Equivalent to:

```perl
print "no alcohol\n"
  if all(5,6,10,12,15,18) < 21;

print "no entry\n"
  if all(5,6,10,12,15,18) < 18;

print "underage\n"
  if any(5,6,10,12,15,18) < 18;
```

• First operand determines the kind of superpositional result of an operation
• Rationale is seen more clearly in comparisons

• In English, adjectives *any* and *all* are left-dominant…

> *Any of us could outthink all of them.*

> *All of us could outthink any of them.*

> *Are all of the results less than any of the thresholds?*

> *Are any of the results less than all of the thresholds?*
if (all(@results) < any(@thresholds) {  
   print "Bad average outcome!"
}

if (any(@results) < all(@thresholds) {  
   print "Bad worst-case outcome!"
}

• Left operand dominance makes quantum comparisons conform to English logic

Super(com)positions

• States of a superposition may be any kind of scalar value:
  - number
  - string
  - reference

• States of a superposition may be any kind of scalar value:

```perl
$wanted = any("Mr","Ms") . any(@names);

print "Reward!
   if $name eq $wanted;

my $huge =
   all( BigNum->new($centillion),
        BigNum->new($googol),
        BigNum->new($SkewesNum)
   );

@humongous = grep {$_ > $huge} @nums;

my $okay =
   all(\&check1,\&check2,\&check3);

die unless $okay->();
• A superposition is a scalar...
• ...so superpositions may have states that are themselves superpositions
$ideal =
   any(
      all("tall", "rich", "handsome"),
      all("rich", "old"),
      all("smart","Australian","rich")
   );

• Operations on a composite superposition operate on each its states...
• ...recursively
• ...in parallel
• For example:

   while (<>)
      my $features = any(split);
      print "True love\n"
      if $features eq $ideal;
   }

Eigenstates again
• Useful to know which states a given scalar superposition has
• For example, a factoring calculation might produce:
   all(76543, 76543, 76543)
• Indicates that computed prime factor was definitely 4321.
• Likewise, a password-checking algorithm might use a disjunctive superposition of unacceptable passwords:
   any("gödel", "escher", "bart")
• May want to list these for the user
• Of course, factoring algorithm might return:
   all(43, 76543, 109)
• No single number equates to all these states
• Represents a failure to factor
• Shor's algorithm can fail in this way
• Hence the actual states of a superposition are rarely useful in themselves.
• What is useful is the list of eigenstates of the superposition
• The (non-superimposed) scalar values to which the superposition may successfully collapse
• List of values @ev for a given superposition $s$ such that: any(@ev) == $s$

• Or perhaps:
  any(@ev) eq $s$

• Use the eigenstate operator
• Applied to superposition...
• ...produces list of “attainable” states

  print "The factor was: ",
  eigenstates($factors);

  print "Don't use any of these: ",
  eigenstates $badpasswds;

• Conjunctive superposition typically has no eigenstates...
• ...unless all its states are equivalent
• That’s usually only when its states are identical
• But may also happen if the states are disjunctive superpositions
• For example:

  all(
    any("tall", "rich", "handsome"),
    any("rich", "old"),
    any("smart","Australian","rich")
  );

• Has a single eigenstate: "rich"

Discrete Truths
• Boolean logic is implemented by binary operators
• Fundamentally the same as arithmetic operators (but lower precedence)
• Examples so far assume the same kind of parallel computations
• Comparison of two superpositions produces a superposition of true's and false's
• Cross-product of comparisons between individual states of two operands.
• But we can do better
• Useful information is lost in conversion to Boolean values
• Namely, which states were responsible for the success of the comparison
• For example:
  
  ```perl
  if (any(@newnums) < @all(oldnums)) {
    print "New depths plumbed";
  }
  ```

  • Tells us if there's a new mimimum in town...
  • ...but doesn't reveal what it is!

  • Could define different semantics for logical operations between superpositions
  • Preserve the intuitive logic of comparisons
  • But also give access to the states that cause success
  • Previously comparing two superpositions yielded a superposition of compared states
  • Now it will return a superposition of eigenstates that caused comparison to be true

  • For example:

  ```plaintext
  all(7,8,9) <= any(7,8,9)   # A
  all(5,6,7) <= any(7,8,9)   # B
  any(6,7,8) <= all(7,8,9)   # C
  ```

  • Previously yielded:

  ```plaintext
  all(1,1,1,0,1,0,0,1)       # A
  all(1,1,1,1,1,1,1,1)       # B
  any(1,1,1,1,1,0,1,1)       # C
  ```

  • Will now yield:

  ```plaintext
  all()                      # A
  all(5,6,7)                 # B
  any(6,7)                   # C
  ```

  • Success of comparison (truth of result) no longer determined by values of the resulting states
  • Now determined by the number of states in the resulting superposition
  • No states means none of the parallel state comparisons succeeded
  • So comparison must have failed
  • One or more states means at least one parallel comparison succeeded
  • Comparison as a whole succeeded...
  • ...if the result is disjunctive
• If the result is conjunctive, comparison may succeed...

• ...but only if result has as many states as original left operand

• Unfortunately result loses touch with left operand once comparison is complete

• So conjunctive comparisons must determine truth of their result during the operation

• Then set an internal flag in the result indicating whether it is true

• Now it’s possible to check a comparison and determine what made it succeed:

  ```perl
  $newmins =
  any(@newnums) < all(@oldnums);

  if ($newmins) {
      print "New depths plumbed: ",
          eigenstates $newmins;
  }
  ```

• Implies we can conduct searches for minima and maxima in constant-time:

  ```perl
  sub min {
      eigenstates any(@_) <= all(@_)
  }

  sub max {
      eigenstates any(@_) >= all(@_)
  }
  ```

**String Theory**

• How do we stringify a superposition?

• Don’t want them to be cumbersome

• Want the most compact form…

• ...that’s equivalent to the original superposition

• If the superposition will always collapse to a single eigenstate, just represent it by that eigenstate

  ```perl
  any("Connor", "Connor");

  all("Connor", "Connor");

  any( all("Connor", "Connor"),
       all("Connor", "Ramirez")
  );
  ```
all( any("Kurgan", "Connor"),
    any("Connor","Ramirez")
    any("Wyatt","Connor")
    );

• All stringify to "Connor"
Now no need to invoke eigenstates for unique results:

    print "alpha: ",
    any(@words) le all(@words);

    print "omega: ",
    any(@words) ge all(@words);

• But what if not exactly one eigenstate?
• Disjunctive superposition becomes "any(eigenstates)":

    any( all("You", "You"),
        all("Me", "Him"),
        all("Her", "Her")
    );

• Represented by the string:

    "any(You,Her)"

• Disjunctive superposition becomes "any(eigenstates)":

    any( all("You", "You"),
        all("Me", "Him"),
        all("Her", "Her")
    );

• Represented by the string:

    "any(You,Her)"

• Conjunctive superposition becomes "all(stringified_states)"

    all( any("You", "You"),
        any("Me", "Him"),
        any("Her", "Her")
    );

• Represented by the string:

    "all(You,any(Me,Him),Her)"
Superpositive Numbers

• What about numerification?
• Again, if the superposition has single eigenstate, use that
• For example, array element at smallest index:
  
  ```
  print "The smallest element is: ",
  $array[any(@index)<=all(@index)];
  ```

• Same effect as:
  
  ```
  print "The smallest element is: ",
  $array[(sort(@index))[0]];}
  ```

• But superposition is constant-time!
• Suppose superposition has no eigenstates?
• No numerical value to which it could collapse
• So result is undefined
• Would then be autoconverted to zero (with a warning)
• What if disjunctive superposition has more than one eigenstate?
• Could collapse to any of those values
• So we let it do exactly that
• (Pseudo-)randomly select one eigenvalue as numeric equivalent
• Conveniently provides random selection from a list:
  
  ```
  print "And the lucky winner is:",
  $entrant[any(@entrant)];
  ```

• These probabilistic semantics could also have been used to stringify
• That is, don’t convert to "any(eigenstates)"
• Instead, select one state at random:
  
  ```
  print "And the lucky winner is:",
  any(@entrant);
  ```

A prime example

• Let’s return to what the NSA would call “da scene of da crime”
• Prime numbers
• Factorization
• Anticryptography

• Here’s a constant-time prime-number tester:

```perl
sub is_prime {
    @_[0] % all(2..sqrt(@_[0])+1) != 0
}
```

• This is actually over-engineered

• Superpositions carry out operations in parallel
• So it doesn’t matter how many divisors we use
• This is marginally faster:

```perl
sub is_prime {
    @_[0] % all(2..$_[0]-1) != 0
}
```

• Of course, `is_prime` takes a single scalar argument
• So we could pass it a superposition
• Here’s a constant-time filter for detecting twin primes:

```perl
sub has_twin {
    is_prime(@_[0]) &&
    is_prime(@_[0]+any(+2,-2));
}
```

• Can’t code it in a single call to `is_prime`:

```perl
sub has_twin {
    my $values =
        all(@_[0],@_[0]+any(+2,-2));
    return is_prime($values);
}
```

• Conjunctive superpositions interfere in modulus of `is_prime`
• Factoring numbers also trivial using superpositions
• The factors of N are those quotients q of N/n (0 < n < N) that are also integral.
• q is integral if \[ q \]

```perl
sub factors {
    my $q = @_[0]/any(2..$_[0]-1);
    return eigenstates(int($q)==$q);
}
```

• Can use `factors` to compute GCD of two numbers
• Costs only a single extra comparison
• GCD is the largest factor two numbers share

• Use factors to find the factors of each number

• Find shared factors by forming conjunctive superposition of disjunctive superpositions of individual factor lists

  \[
  \text{all}( \text{any}(\text{factors} \: \$x), \\
  \text{any}(\text{factors} \: \$y) )
  \]

• The eigenstates of this composite superposition must be eigenstates of both the factor lists

• Largest such factor can be extracted by maximizing comparison

  ```perl
  sub gcd {
    my ($x, $y) = @_; 

    my $common = 
      all( any(factors($x)), 
          any(factors($y)) );

    return any(eigenstates $common) 
      >= all(eigenstates $common);
  }
  ```

**Quantum::Superpositions**

• The Quantum::Superpositions module really exists

• Of course, it can't actually superimpose scalar values

• Nor perform parallel operations on them in constant time

• But it *can* simulate the behaviour of superimposed scalars

• Will be upgraded to the real thing…

• …just as soon as Perl is ported to run on a pair of coherent, quantum-entangled, laser-cooled ions

• Meanwhile, the simulation has a number of interesting tricks to teach us

**Quantum OO**

• Superpositions are modelled by objects of three classes:
  – Quantum::Superpositions::Disj
  – Quantum::Superpositions::Conj
  – Quantum::Superpositions::Conj::True

• Hereafter QSD, QSC, and QSCT

• Derive from a common base class: Quantum::Superpositions (QS)
• And QSCT derived from QSC

• QS objects are blessed arrays in which the superposition’s individual states are stored

• The all and any operators are just constructors for QSC and QSD objects respectively

```
package Quantum::Superpositions;

sub new {
    my $class = shift;
    bless [@_], $class;
}

sub any { Quantum::Superpositions::Disj->new(@_) }

sub all { Quantum::Superpositions::Conj->new(@_) }

sub stringify { ... }

sub boolify { ... }

# etc.
```

• Also need an all_true constructor to produce QSCT objects

```
sub all_true {
    Quantum::Superpositions::Conj::True->new(@_)
}
```

• Each of the two derived classes is a Gammarian facade

• Adds no new attributes to the base class

• Just provides a means of polymorphically specifying appropriate operator and type conversion semantics

```
package Quantum::Superpositions::Disj;
use base 'Quantum::Superpositions';

sub stringify { ... }
```
• But binary operator semantics depend on both operands
• So they must be multiply dispatched
• To simplify that, use Class::Multimethods module
• More on that shortly

**Eigenstates**

• The eigenstates operator is functionally identical for conjunctive and disjunctive superpositions
• So it’s implemented as a method in the QS base class

```perl
sub eigenstates($) { 
    my ($self) = @_; 

    my %uniq; 
    my @states = collapse($self); 
    @uniq{@states} = (); 

    use warnings 'numeric'; 
    return grep { 
        my $okay=1; 
        local $SIG{__WARN__} = sub {okay=0}; 
        istrue($self eq $_) || 
        istrue($self == $_) && $okay 
        } keys %uniq; 
}
```

• Derives a list of candidate states (by calling collapse)
• Values in resulting list then compared for equality with the original superposition
• Any for which the equality holds are returned as the eigenstates
• Comparison needs to try both == and eq
• But how to avoid "Argument isn't numeric in eq..." message?
• Trap it...remember it...ignore it...recall it...test it...condemn it

sub eigenstates($) {
  my ($self) = @_;

  my %uniq;
  my @states = collapse($self);
  @uniq{@states} = ();

  use warnings 'numeric';
  return grep {
    my $okay=1;
    local $SIG{__WARN__} = sub { $okay=0 };  
    istrue($self eq $_) || 
    istrue($self == $_) && $okay 
  } keys %uniq;
}

• Full superpositional equality tests are expensive
• Have to test all pairwise combinations of states
• Wasteful 50% of the time
• For disjunctive superposition value matches if *any* state matches
• For conjunctive superposition match fails if *any* state doesn’t match

• Optimise equality using multimethod *istrue*
• Short-circuits to *true* on any matching state in a disjunctive superposition
• Short-circuits to *false* on any failure to match a conjunctive superposition’s state

**Type conversions**
• Boolification and numerification semantically identical for both kinds of superposition
• Hence implemented as methods of the base class
• String conversion differs for each kind of superposition
• Implemented as separate polymorphic methods in derived classes.

**Boolification**
• Trivial
• Superposition true if it has one or more eigenstates
• Or if it’s been marked explicitly as true
• That is, if its class is QSCT

```perl
package Quantum::Superpositions;
sub boolify { eigenstates($_[0]) ? 1 : 0 }
```

```perl
package Quantum::Superpositions::Conj::True;
use base 'Quantum::Superpositions::Conj';
sub boolify { 1 }
```

**Numerification**

• Numeric conversion equally straightforward
• Compute eigenstates of superposition
• If none, return `undef`
• Otherwise, choose one eigenstate at random and return it

```perl
package Quantum::Superpositions;
sub numerify {
    my @states = eigenstates($_[0]);
    return $states[0] unless @states > 1;
    return $states[rand @states];
}
```

```perl
package Quantum::Superpositions;
sub numerify {
    my @states = eigenstates($_[0]);
    return $states[rand @states];
}
```

**Stringification**

• Stringifiers use eigenstates too
• If exactly one, that’s the string representation
• Otherwise QSC concatenates states, and interpolates into "all(...)"
• QSD concatenates eigenstates and interpolates into "any(...)"
package Quantum::Superpositions::Disj;
use base 'Quantum::Superpositions';

sub stringify {
    my @eigenstates = eigenstates $_[0];
    return "@eigenstates" if @eigenstates == 1;
    return "any(" . join("","",@eigenstates).")"
}

package Quantum::Superpositions::Conj;
use base 'Quantum::Superpositions';

sub stringify {
    my ($eigenstate) = eigenstates $_[0];
    return "$eigenstate" if $eigenstate;
    return "all(" . join("","", @{$_[0]}).")"
}

The eigencache
• So eigenstates get used on every conversion
• Might be worth caching.
• But no room in the individual superposition objects
• May have to be cached in a lexical hash variable

my %eigencache;
sub eigenstates($) {
    my ($self) = @_;
    my $eigencache_id = overload::StrVal($self);
    return @{$eigencache{$eigencache_id}}
        if defined $eigencache{$eigencache_id};
    my %uniq;
    @uniq{collapse($self)} = ();
    use warnings 'numeric';
    return @{$eigencache{$eigencache_id}} = grep {
        my $okay=1;
        local $SIG{__WARN__} = sub { $okay=0 };  
        istrue($self eq $_) ||
        istrue($self == $_) && $okay
    } keys %uniq;
}
• Careful with that destructor, Eugene!

    package Quantum::Superpositions;

    sub DESTROY {
        delete $eigencache{overload::StrVal($_[0])}
    }

Unary Operations

• Unary operations can be handled by a single method: quop
• Two rules for unary arithmetic operations on superpositions...
• In both cases, recursively apply operator to each state
• Only difference is the class of the resulting value

    package Quantum::Superpositions::Conj;

    sub quop {
        my ($self, $op) = @_;
        return all map { $op->($_) } @{$self};
    }

    package Quantum::Superpositions::Disj;

    sub quop {
        my ($self, $op) = @_;  
        return any map { $op->($_) } @{$self};
    }

    package Quantum::Superpositions;

    sub quop {
        my ($self, $op) = @_;  
        return bless [map { $op->($_) } @{$self}], ref $self;
    }

• Logical unary operations similar
• Implemented by method: qulop
package Quantum::Superpositions::Conj;

sub qulop {
    my ($self, $op) = @_;
    return all grep { $op->($_) } @{$self};
};

package Quantum::Superpositions::Disj;

sub qulop {
    my ($self, $op) = @_;  
    return any grep { $op->($_) } @{$self};
};

package Quantum::Superpositions;

sub qulop {
    my ($self, $op) = @_;  
    return bless [grep { $op->($_) } @{$self}], ref $self;
};

Binary Operations

• Binary operations depend on the nature of both arguments

• So implemented as generic multimethods: qbop and qblop

• Hope you all read my article in the latest Perl Journal...

• The qbop multimethod has nine variants

• Cover the eight rules of binary superpositional arithmetic...

• ...plus the special case of interactions between non-superimposed scalars

use Class::Multimethods;

multimethod qbop => ( QSC, QSC, CODE ) => sub {
    all map { qbop(@$_, $_[2]) } cross($_[0], $_[1]);
};

multimethod qbop => ( QSD, QSD, CODE ) => sub {
    any map { qbop(@$_, $_[2]) } cross($_[0], $_[1]);
};

multimethod qbop => ( QSC, QSD, CODE ) => sub {
    all map { qbop($_, $_[1], $_[2]) } @{$_[0]};
};
multimethod qbop => ( QSD, QSC, CODE ) => sub {
    any map { qbop($_, $_[1], $_[2]) } @{$_[0]};
};

multimethod qbop => ( QSC, '$', CODE ) => sub {
    all map { qbop($_, $_[1], $_[2]) } @{$_[0]};
};

multimethod qbop => ( QSD, '$', CODE ) => sub {
    any map { qbop($_, $_[1], $_[2]) } @{$_[0]};
};

multimethod qbop => ( '$', QSC, CODE ) => sub {
    all map { qbop($_, $_[1], $_[2]) } @{$_[1]};
};

multimethod qbop => ( '$', QSD, CODE ) => sub {
    any map { qbop($_, $_[1], $_[2]) } @{$_[1]};
};

multimethod qbop => ( '$', '$', CODE ) => sub {
    return $_[2]->(@_[0..1]);
};

Logical Operations

• The qblop multimethod also has nine variants

• Different from qbop

• All return a superposition of the states of the first operand...

• ...for which the generic operation is true

multimethod qblop => ( QSC, QSC, CODE ) => sub {
    my ($leftop, $rightop, $op) = @_; 
    return all() unless @{$leftop} && @{$rightop};
    istrue(qlop($leftop, $rightop, $op)) || return all()
    foreach cross($leftop,$rightop);
    return all_true @{$leftop};
};

multimethod qblop => ( QSD, QSD, CODE ) => sub {
    my ($leftop, $rightop, $op) = @_; 
    return any() unless @{$leftop} && @{$rightop};
    return any grep { istrue(qlop($leftop, $rightop, $op)) } @{$rightop};
};
multimethod qblop => ( QSC, QSD, CODE ) => sub {
    my ($left, $right, $op) = @_;  
    return all() unless @{$leftop} && @{$rightop};  
    my @conj_states = @{$leftop};  
    my @disj_states = @{$rightop};  
    my $matches = 0;  
    foreach my $conj ( @conj_states ) {  
        foreach my $disj ( @disj_states ) { 
            ++$matches && last  
            if istrue(qlop($conj, $disj, $op));  
        }  
    }  
    return all() unless $matches == @conj_states;  
    return all_true @{$leftop};  
};

Overloading

• Generic operator implementations and conversion methods then mapped onto corresponding operators

• Use standard Perl operator overloading mechanism

use overload
    q{+}    =>  sub {qbop(swap(@_), sub{$_[0]+$_[1]})},
    q{*}    =>  sub {qbop(swap(@_), sub{$_[0]*$_[1]})},
    q{%}    =>  sub {qbop(swap(@_), sub{$_[0]/$_[1]})},
    q{<}    =>  sub {qblop(swap(@_), sub{$_[0]<$_[1]})},
    q{==}   =>  sub {qblop(swap(@_), sub{$_[0]==$_[1]})},
    q{!=}   =>  sub {qblop(swap(@_), sub{$_[0]!=$_[1]})},
    q{sqrt} =>  sub {$_[0]->quop( sub{sqrt $_[0]} )},
    q{neg}  =>  sub {$_[0]->quop( sub{-$_[0]} )},
    q{~}    =>  sub {$_[0]->quop( sub{~$_[0]} )},
    q{!!}   =>  sub {$_[0]->quulp( sub{!$_[0]} )},

    # etc.

    q{bool} =>  'boolify',
    q{""}  =>  'stringify',
    q{0+}   =>  'numerify',
    ;
Extensibility

- Recall the quantum factorizing subroutine:

  ```perl
  sub factors {
    my $q = $_[0]/any(2..$_[0]-1);
    return eigenstates(int($q)==$q);
  }
  ```

  - The `int` used isn't the standard, built-in `int`
  - That `int` doesn't DWIM with superpositions
  - Quantum::Superpositions module provides a mechanism to "auto-quantise" built-ins and subroutines
  - To specify that a subroutine or built-in should be sensitive to superpositions:

    ```perl
    use Quantum::Superpositions
    UNARY => ["Core::int"];

    sub factors {
      my $q = $_[0]/any(2..$_[0]-1);
      return eigenstates(int($q)==$q);
    }
    ```

    ```perl
    use Quantum::Superpositions
    BINARY => ["Core::index"];

    print index(any("opts","tops","spot"), "o");
    print index("stop", any("p","s"));
    ```

    ```perl
    use Quantum::Superpositions
    BINARY_LOGICAL => ['main::odder'];

    print odder(any(1234,2468), 666);
    ```
Conclusion

• Qubit-based computing requires deep understanding of:
  – quantum physics,
  – tensor mathematics
  – nuclear engineering
  – molecular chemistry
  – algorithmic design
  – Murphy’s Law

• Unlikely to ever be accessible to most of us JAPHs

• Superposition-based computing offers similar power

• ...but requires only a small, intuitive extension to Perl