Quantum logic in Perl and the effect it has on organized gambling

Quantum Casino

Superpositions from the realm of quantum physics will soon be available as so-called junctions in the Perl 6 core. A module is already available. This month's article takes a lighthearted look at this revolutionary concept based on a script that plays blackjack. BY MICHAEL SCHILLI

Just recently I was browsing through the books in my local bookshop when I noticed a paperback called "Winning Casino Play" [2], where a professional gambler explained how to play the Casinos in Las Vegas and Atlantic City with a fair chance of winning. Blackjack was one of the subjects the author covered in detail as it is one of the games that gives you the fairest odds on the Casino scene.

19, 20, 21 ... busted!

I decided to put in some practice before trying my luck at the Casino, and set out to write a Perl script that would simulate the game. The rules are as follows: the game is played between the player and the dealer. Several packs of 52 cards are used. Cards are dealt singly from the so-called "shoe", or card holder.

The object of the game is to draw as many cards as possible to have a hand of exactly 21. But be careful – if you go over

Listing 1: blackjack

```
#!/usr/bin/perl
##################################
# play - Blackjack against Las Vegas Dealer
# Mike Schilli, 2003 (m@perlmeister.com)
##################################
use warnings; use strict;
use Blackjack;
use Term::ANSIColor qw(:constants);
use Term::ReadKey;
$| = 1; my $total = 0;
my $shoe = Blackjack::Shoe->new(
  nof_decks => 4);
{
 if($shoe->remaining() < 52) {
    print "Shuffling ...\n";
    $shoe->reshuffle();
 }
my $player = Blackjack::Hand->new(
   shoe => $shoe);
 my $dealer = Blackjack::Hand->new(
             shoe => $shoe);
  $dealer->draw();
  P(RED, "D", $dealer);
  $dealer->draw();
  $player->draw();
  $player->draw();
  while(!$player->busted()) {
    P(BLUE, "P", $player);
    print "([H]it/[S]tand/[Q]uit) ";
    ReadMode 4;
    my $move = ReadKey(0);
    ReadMode 0;
    print "\r";
    last if $move =~ /^s/i;
    exit 0 if $move =~ /^q/i;
    $player->draw();
  }
  P(BLUE, "P", $player);
  while(!$dealer->busted() and
        $dealer->count("soft") < 17) {
      P(RED, "D", $dealer);
      $dealer->draw();
  }
    P(RED,  "D", $dealer);
    $total += $player->score($dealer);
    print "Score: ",
        $player->score($dealer),
        ", Total: ", $total, "\n\n";
    redo;
 }
 sub P { # Print status in color
      print(BOLD, $_[0], "$_[1]", "[",
      $_[2]->count_as_string(), "]",
      RESET, ": ", $_[2]->as_string(), "\n")
 }
```

21, you automatically lose. This is known as being busted. The cards from 2 through 10 are valued as indicated, and the face cards (Jack, Queen, King) are all valued at ten points. An Ace can count as either 1 or 11.

Watch out for Superposition!

1 *or* 11? Right, if you draw a 7, an 8 and an Ace, $7+8+11=26$ would bust you. Instead you can stay in the game by counting the Ace as 1: $7+8+1=16$.

In other words, a hand does not have a fixed value, but two overlapping states (26,16), of which we can choose the most favorable: 16. Imagine drawing four Aces. This would return four possible states (4,14,24,34). But as 24 or 34 are not exactly winning hands, we are only interested in two of them: 4, the soft count, and 14, the so-called hard count.

Overlapping states are referred to as superpositions in quantum physics, and allow a particle to occupy two spaces at the same time.

After installing the *Quantum::Superpositions* module by Damian Conway (it is available from CPAN), you can create a superposition of the numerical values previously displayed, by calling the *any()* function:

```
use Quantum::Superpositions;
my $count = any(4,14,24,34);
```

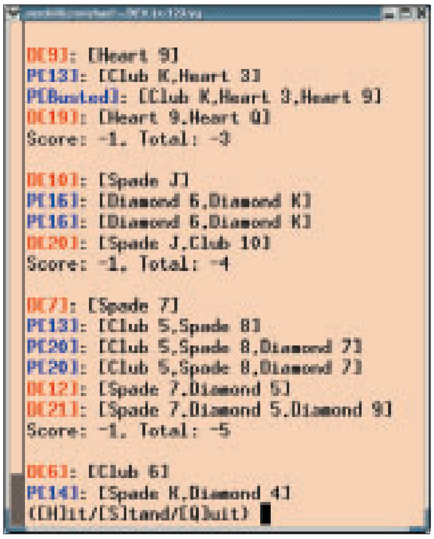

Figure 1: A game of blackjack in the command line

From this point on, the *$count* variable appears to have four different values. A seemingly insane logical expression, such as

```
if($count == 4 and
   $count == 14) {
     print "Right!<\\>n";
}
```

will return the value true and execute the *print* command. Also, logical comparisons such as *$count* < = *21* are no longer true or false in the context of *Quantum::Superpositions*, but return a superposition of the states that match the condition. The following syntax will filter the irrelevant states above 21 from the possible results which depend on how we count the Ace (4,14,24,34):

```
$count = ($count <= 21);
```

Now *$count* contains only *any(4,14)*. That saves a lot of typing! Arithmetic operations based on superpositions are also quite simple to formulate as *Quan-*

Listing 2: Blackjack.pm

```
################################
# Blackjack.pm
# Mike Schilli, 2003 (m@perlmeister.com)
################################
use warnings; use strict;
#===============================
package Blackjack::Shoe; #===
#===============================
use Algorithm::GenerateSequence;
use Algorithm::Numerical::Shuffle
     qw(shuffle);
################################
sub new {
################################
   my($class, @options) = @_;
   my $self = {nof_decks => 1, @options};
   bless $self, $class;
   $self->{cards} = $self->reshuffle();
   return $self;
}
################################
sub reshuffle {
################################
   my($self) = @_;
   my @cards =
(Algorithm::GenerateSequence->new(
   [qw( Heart Diamond Spade Club )],
   [qw( A 2 3 4 5 6 7 8 9 10 J Q K )])
     ->as_list()) x $self->{nof_decks};
   return shuffle \@cards;
}
################################
sub remaining {
################################
    my($self) = @_;
    return scalar @{$self->{cards}};
}
################################
sub draw_card {
################################
    my($self) = @_;
    return shift @{$self->{cards}};
}
#===============================
package Blackjack::Hand; #===
#===============================
use Quantum::Superpositions;
################################
sub new {
################################
   my($class, @options) = @_;
   my $self = { cards => [], @options };
   die "No shoe" if !exists $self->{shoe};
   bless $self, $class;
}
################################
sub draw {
################################
 my($self) = @_;
 push @{$self->{cards}},
   $self->{shoe}->draw_card();
}
################################
sub count {
################################
   my($self, $how) = @_;
   my $counts = any(0);
   for(@{$self->{cards}}) {
     if($_->[1] =~ /\d/) {
       $counts += $_->[1];
     } elsif($_->[1] eq 'A') {
```

tum::Superpositions will overload all operators. If the value *any(4,14)* is store in *$counts*,

```
$counts += 10;
```

will result in *any(14,24)*.

Thus, *any()* specifies a so-called disjunctive superposition that can be queried for all of its states, and will respond correctly to logical queries for any of those states.

The second function introduced by *Quantum::Superpositions*, *all()*, defines a conjunctive superposition that has all the available states simultaneously. The following condition is *not* true:

```
my $count = all(4,14,24,34);

if($count <= 21) {
    print "None busted<\\>n";
}
```

as not all of the states contain values of less than 21. In contrast, *all(4,14) < = 21* would be true.

Is the Cat Dead?

If a researcher tests the truths of quantum physics in the real world, multiple states will collapse to a single state, removing any ambiguities: This means that Schrödingers cat [3] must either be dead as a dodo or still alive.

This is different in Perl: You can play around with ambiguous states without destroying the system. To discover the states a superposition contains, *Quantum::Superpositions* imports the *ownstates()* function, which simply returns a list of states:

```
my @counts = ownstates($count);
```

These three functions *any()*, *all()* and *ownstates()* allow a developer to build breathtaking program constructs. The following syntax is all you need to discover the *soft count* for the specified cards, that is the minimum from *any(4,14,24,34)*:

```
my $counts = any(4,14,24,34);
my $soft =
  ($counts <= all(ownstates
($counts));
```

as the logical comparison uses *$counts* to return a superposition of all states that are less than or equal to *all* states of the superposition – this is the classical definition of a minimum.

Perl Card Shoe Class

The implementation: The *Blackjack.pm* listing contains two classes: *Blackjack::Shoe*, which represents the card shoe from which the dealer removes the cards to be dealt up, and *Blackjack::Hand*, which represents the player's or dealer's hand.

The card shoe uses the CPAN *Algorithm::GenerateSequence* module to generate a few packs of cards. The *new()* constructor accepts references to an array, whose elements it combines. If the first array contains all the suits (Heart/Diamond/Spade/Club) and the second all the values (A 2 3 4 5 6 7 8 9 10 J Q K) of a card game, the *as_list()* method will return a list of combinations

Listing 2: Blackjack.pm

```
     $counts = any($counts+1,
         $counts+11);
    } else {
     $counts += 10;
    }
  }
        # Delete busted hands
 $counts = ($counts <= 21);
        # Busted!!
 return undef if !
ownstates($counts);
 return $counts unless
defined $how;
 if($how eq "hard") {
   # Return minimum
  return int($counts <=
    all(ownstates($counts)));
 } elsif($how eq "soft") {
   # Return maximum
  return int($counts >=
    all(ownstates($counts)));
 }
}
################################
sub blackjack {
################################
    my($self) = @_;
    my $c = $self->count();
    return 1 if $c == 21 and
$c == 11 and
       @{$self->{cards}} == 2;
    return 0;
}
################################
sub as_string {
################################
   my($self) = @_;
   return "[" . join(',',
map({ "@$_" }
      @{$self->{cards}})) .
"]";
}
################################
sub count_as_string {
################################
    my($self) = @_;
    return $self->busted() ?
     "Busted" : $self-
>blackjack() ?
     "Blackjack" : $self-
>count("soft");
}
################################
sub busted {
################################
    my($self) = @_;
    return ! defined $self-
>count();
}
################################
sub score {
################################
    my($self, $dealer) = @_;
    return -1 if $self-
>busted();
    return 1 if $dealer-
>busted();
  return 0 if $self-
>blackjack() and
         $dealer->blackjack();
  return 1.5 if $self-
>blackjack();
  return -1 if $dealer-
>blackjack();
  return $self->count("soft")
<=>
       $dealer->count("soft");
}
1;
```

that map to individual cards:: Heart A, Heart 2, … Club Q, Club K. The list that this creates is replicated by the *x* operator to reflect the number of card games to be placed in the shoe (line 29).

The *Algorithm::Numerical::Shuffle* module finally exports the *shuffle* method. This shuffles the elements of an array passed to it, using the Fisher-Yates method. The *reshuffle()* method of the *Blackjack::Shoe* object places the number of 52 card decks defined in the instance variable *nof_decks* into the card holding shoe.

remaining() returns the number of cards left in the shoe; this allows the dealer to check that the game can be completed with the remaining decks. *draw_card()* draws a card from the shoe and returns it as a reference to an array, whose first element is the suit (Heart, Diamond, Spade, Club) and whose second element is the face value (A, 2, 3, …, J, Q, K).

The *Blackjack::Hand* class represents the principle of a player holding a hand of cards. It has to do with drawing cards from the shoe and their values – it does not matter whether the hand belongs to the player or the dealer. The constructor

```
Blackjack::Hand->new(shoe =>
$shoe)
```

links the player/dealer to the card shoe from which the new cards will be drawn. The *draw()* method adds one card to a hand.

Using Superpositions to Count the Score

The *count()* method in line 64 ff. counts the score for a hand, and returns the result either as a superposition, a hard count (*$hand->count("hard")*), or a soft count *$hand->count("soft")*.

To allow this, the *for* loop in line 68 ff. iterates through the cards, checking the individual values and adding them. The *$count* variable stores the superposition of potential hand scores. It makes use of the fact that *Quantum::Superpositions* overloads the + operator, so that *$counts += 10* will add 10 to all superpositions in *$counts*. When an Ace is drawn, the number of superpositions is doubled with 1 being added to half of them, and 11 to the other half.

```
$counts = any($counts+1,
$counts+11);
```

Line 79 immediately removes any scores above 21 from the superpositions. If a superposition no longer contains a value after this operation, that is *ownstates($counts)* returns an empty list, the player has obviously exceeded the maximum score of 21 and the hand is busted. The *count()* method in line 83 ff. discovers the hard and soft counts, using the tricks discussed previously to find the minimum and maximum values. The *int()* function then strips the superposition of its special traits and makes it scalar.

Blackjack Wins

If a player's hand contains exactly one Ace and one 10 point card, this constitutes a Blackjack, and trumps any other hands with 21 points. The *blackjack()* method defined in line 94 ff. checks for this situation by checking whether the value of the hand corresponds to a superposition with the values 11 and 21 and if the hand contains exactly two cards.

The *score()* method ascertains whether a hand has won or lost against the dealer's *Blackjack::Hand* object, which it accepts as a parameter. If the result is negative, the player loses. If it is positive, the player wins. And now let's look at a few special cases: The player's Blackjack pays 1:1.5, while the dealer only gets straight odds for the player's bet. If the player has more than 21, the hand is busted, and the player loses, even if the dealer also exceeds 21 later.

Text Color and Keyboard Hits

The *blackjack* listing contains a script that allows you to play a computerized dealer at Blackjack, just like in Las Vegas. It uses the classes defined in *Blackjack.pm* for the dealer's card shoe, and the player's and dealer's hands. Text::ANSIColor is used to provide some colored output; the module exports constants, such as *BOLD*, *RED*, *BLUE* or *RESET*, which are tagged onto terminal escape sequences, based on the constant values passed to it as *:constants*. This allows for bold or colored output, or will return output back to normal display mode.

Text::ReadKey in *Raw Mode* (initiated by *ReadMode 4*), allows you to capture values for key-presses, without the user having to hit the *Enter* key. Toggling to *ReadMode 0* then returns the terminal to *Cooked Mode*, where input must consist of whole lines, if the user hits *Enter* and none of the keys pressed so far have provoked a reaction from the program.

When it is the users turn, the following output is shown on screen:

```
[H]it/[S]tand/[Q]uit
```

The player can opt to *Hit* (draw another card), *Stand* (not draw any more cards), and *Quit* (the game), by pressing the *H*, *S* or *Q* keys.

Figure 1 shows a typical game. The dealer starts the game by drawing two cards, although only one card will be shown face up. The player is then dealt two cards and can ask for another card, to add points to her hand. If the player opts to stand, the computerized dealer will then follow a strict pattern for its own hand. If the total *soft count* is less than 17, the dealer draws a new card. 17 or more causes the dealer to stand, this is not influenced by the player's hand, and then the dealer has to either pay out or collect.

Feel free to use Blackjack.pm. You might like to add a GUI, or a write a TCP/IP server to play Blackjack across the network. Have fun, and as they say in Las Vegas, "Good Luck!" ■

INFO

[1] Listing for this article: *ftp://www.linux-magazin.de/pub/listings/magazin/2003/12/Perl*

[2] Avery Cardoza, "Winning Casino Play", Cardoza Publishing, 3rd Ed., 2003, 1-58042-090-7

[3] Illustration of Schrödingers experiment with his cat: *http://mist.npl.washington.edu/npl/int_rep/tiqm/TI_fig_09.html*

THE AUTHOR

Michael Schilli works as a Web engineer for AOL/Netscape in Mountain View, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.