The main topic for today is **data structures on demand**, by means of **program transformations** that guarantee to preserve correctness:

• In languages such as Haskell, data structures are very easy to use (lists and tuples built-in) and define (recursive data types). Introduction to Perl: Eighth Lecture • In languages like C, building data structures seems hard (which is why you should build a toolkit), so you tend to only build data Duncan C. White (d.white@imperial.ac.uk) structures for the macro-scale. Dept of Computing. Imperial College London January 2015 then wrap up the course. Introduction to Perl: Eighth Lecture January 2015 Data Structures On Demand Finding Unique Elements • While writing a single function, you often write code that computes a single answer. Sometimes you can transform this code by building a data structure enabling you to *lookup all* my @uniq; foreach my \$i (0..\$#array) answers of that kind. • For example, given an unsorted array of numbers: mv \$count = 0; @array = ( 17, 5, 3, 17, 2, 5, 7, 6, 6, 10, 3 ); • Consider finding unique values from such an array. We might write the following naive code (**eg1**): 3 # build @uniq, an array of all unique elements of @array my @uniq; foreach my \$i (0..\$#array) # foreach index i in @array # count how many elements  $\operatorname{array}[j]$  (i!=j) are the same as  $\operatorname{array}[i]$ # build @uniq, an array of all unique elements of @array mv \$count = 0: foreach my \$j (0..\$#array) my @uniq; foreach my \$x (@array) \$count++ if \$i != \$j && \$array[\$i] == \$array[\$j]; # unique if \$count == 0 my count = 0;push @uniq, \$array[\$i] if \$count == 0; 3

• This is very C-style code! index based, unclear, 13 lines long, could harbour bugs. Worse still, it's  $O(N^2)$ .

```
January 2015
              3 / 24
```

• In Perl, data structures are even easier to use than in Haskell - so

- simple that building optimal data structures and changing them when you change your mind - becomes a useful programming technique in it's own right.
- I call this the **Agile Data Structures** approach.

We'll also talk briefly about testing, benchmarking and profiling, and

Introduction to Perl: Eighth Lecture

January 2015

## Data Structures On Demand Program Transformations

• Our first transformation is to notice that we can eliminate the  $i_{i} = i_{j}$  test, and compare the count with one not zero (eg2):

```
# build @uniq, an array of all unique elements of @array
                                        # foreach index i in @array
        # how many elements array[j] are the same as array[i] (inclusive)
        foreach my $j (0..$#array)
                $count++ if $array[$i] == $array[$j];
        # unique if $count == 1 (array[i] itself)
        push @uniq, $array[$i] if $count == 1;
```

• Next, notice that we no longer use indices *i* and *j* separately from

array[i] and array[i], so we can now loop over the values (eg3):

Introduction to Perl: Eighth Lecture

```
# how many elements y are the same as x (including x)?
foreach my $y (@array)
       $count++ if $x == $y;
# unique if $count == 1 (x itself)
push @uniq, $x if $count == 1;
```

• Our next transformation is to notice that the inner loop can be replaced with a call to grep (eg4). Recall that grep constructs a list, and assigning that list to a scalar scount delivers the number of elements in the list:

- All the above transformations have improved the clarity of the code, we're much more confident that this is correct now.
   However, still O(N<sup>2</sup>) because grep (and map) count as O(N).
- But now we make a simple observation: Over the course of the foreach loop, we calculate the frequency of *every array element*.
- So why not pre-calculate the element frequencies ahead of time? This suggests a new data structure (a bag or frequency hash): my %freq; # array element -> frequency of that element

```
Duncan White (CSG)
```

```
Introduction to Perl: Eighth Lecture
```

```
Data Structures On Demand Non-unique Elements
```

• Suppose we want an array of the *distinct non-unique* values instead. Non-unique values (ignoring *distinct*) are easy, simply

```
change freq == 1 tO freq > 1:
my %freq; map { $freq{$_}++ } @array;
my @nonuniq = grep { $freq{$_}> 1 } @array;
```

- However, this includes each non-unique element many times.
- For example, if @array = (1,1,1,2,2) then @nonuniq = (1,1,1,2,2) whereas distinct suggests that we wanted @nonuniq = (1,2).
- To remove duplicates from @nonuniq, we can use a standard *turn it into a set and extract the keys* idiom:

```
my %set = map { $_ => 1 } @nonuniq;
@nonuniq = keys %set;
```

- Recall that keys %set delivers the keys in an unpredictable order. We could say sort keys %set, but our code would become O(NlogN).
- An O(N) alternative that delivers the distinct values in the order they were present in the original array is to replace the set of all items in the array with a set of all items seen so far (eg7):

```
my %freq; map { $freq{$_}++ } @array; # build element -> frequency hash
my %seen; # what elements have we already seen?
my @nonuniq = # build distinct non-unique elements
grep { $freq{$_}> 1 && ! $seen{$_+++} @array;
```

January 2015

5 / 24

- To populate %freq we write: my %freq; foreach my \$x (@array) { \$freq{\$x}++; }

   Once we have %freq our code is: my @uniq; foreach my \$x (@array) { push @uniq, \$x if \$freq{\$x} == 1;
   }
- Bringing this all together, this gives eg5, which is clearly O(N)!
- Next, the %freq building code may be more idiomatically written: my %freq; %freq{\$\_}++ for @array; # build array element -> frequency of that element
- Finally, we notice that the main loop is another grep: my @uniq = grep { \$freq{\$\_} == 1 } @array; # build @uniq, all unique elements of @array
- These two lines are the heart of our final clear simple O(N) version eg6. Compare this to our original 13 line O(N<sup>2</sup>) eg1!
- Of course, we had to allocate a modest extra amount of space for the frequency hash. But it's definitely worth it!

```
Duncan White (CSG)
```

incan White

Introduction to Perl: Eighth Lecture

```
January 2015 6 / 24
```

## Data Structures On Demand Changing the Data Structure

- Finally, after building and using "freq, suppose we realised that other parts of the program need to locate all the *positions* in the original array @array at which a specific value appeared.
- We need a different temporary data structure: my %indexlist; # array element -> list of positions in original array
- Recall that the array contains:

```
@array = ( 17, 5, 3, 17, 2, 5, 7, 6, 6, 10, 3 );
```

Our desired %indexlist comprises:

17	=>	[0,	3],	2	=>	[4],
6	=>	[7,	8],	7	=>	[6],
5	=>	[1,	5],	3	=>	[2, 10],

• To build *xindexlist* we might write naive code (eg8):

```
# initialize all 'inner' array refs to [], maybe several times each
foreach my $value (@array)
{
        $indexlist{$value} = [];
}
# can now freely push positions onto @{$indexlist{$value}}
foreach my $index (0..$#array)
{
        my $value = $array[$index];
        my $varef = $indexlist{$value};
        push @$aref, $index;
}
```

Introduction to Perl: Eighth Lecture

• In fact, the first loop is not needed because Perl **auto-vivifies** array and hash references when needed, as this snippet shows:

```
my $ref = undef; @$ref = (1,2,3); print "@$ref\n"';
```

So that gives us:

```
# push positions onto @{$indexlist{$value}} freely
foreach my $index (0..$#array)
{
    my $value = $array[$index];
    my $aref = $indexlist{$value};
    push @$aref, $index;
}
```

• svalue is only used once, fold it in:

```
foreach my $index (0..$#array)
{
    my $aref = $indexlist{$array[$index]};
    push @$aref, $index;
}
```

• Writing the foreach loop as a procedural map, we end up with the following more idiomatic version:

my %indexlist; map { my \$aref = \$indexlist{\$array[\$\_]}; push @\$aref, \$\_ } 0..\$#array;

 If you're happy to push it one stage further, fold saref in too: my %indexlist; map { push @{\$indexlist{\$array[\$\_]}}, \$\_ } 0..\$#array;

```
Duncan White (CSG)
```

Introduction to Perl: Eighth Lecture

Data Structures On Demand Scaling Agile Data Structures Up

• Please note that this technique isn't only appropriate on the small scale - let's scale it up. We said that we were working inside functions, let's make that explicit now:

```
# @uniq = unique_values( @array ):
        Deliver all non-repeated values from Carray
#
        in the SAME ORDER they were present in @array
fun unique_values( @array )
        my %freq; $freq{$_}++ for @array;
                                                  # array element -> frequency
        my @uniq = grep { $freq{$_} == 1 } @array; # @uniq, unique elements
        return Qunia:
3
#
# @nonuniq = distinct_nonunique_values( @array ):
        Deliver all repeated (non-unique) values from @array
       once each (i.e. distinct), in the SAME ORDER as they
       were first found in @array
#
fun distinct_nonunique_values( @array )
ſ
        my %freq; $freq{$_}++ for @array;
                                                # array element -> frequency
        my %seen;
                                                # elements we've already seen
        my @nonuniq = grep
                                                # distinct non-unique elements
               { $freq{$_} > 1 && ! $seen{$_}++ } @array;
        return @nonuniq;
}
```

- Now, given that \$freq{\$v} == @{\$indexlist{\$v}}, ie. \$v's frequency is the length of \$v's position list, do we need to keep %freq?
- A minimalist would remove %freq, to avoid redundancy. Our uniqueness detector would then be:
   my @uniq = grep { @{\$indexlist{\$\_}}} == 1 } @array;
- Personally, I'd keep both and build them together (eg9): my( %indexlist, %freq ); map { %freq(%array[\$\_]}++; push @{\$indexlist{%array[\$\_]}}, \$\_; } 0..\$#array;
- Let's pause for a moment and take stock of what we've done:
  - In a series of very small example programs (each < 20 lines long)...
  - We've shown how to gradually transform low level algorithmic code, into shorter, clearer, more obviously correct code...
  - Using temporary data structures (*scaffolding*) and higher-order functions such as grep and map...
  - To make the original problem much easier to solve..
  - Sometimes even making the code faster and more efficient.
- This is a sufficiently rare combination of good characteristics that it's worth celebrating, noting that it's only possible because Perl makes building optimal data structures so simple.

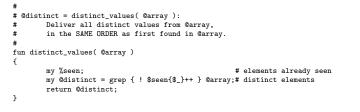
```
Duncan White (CSG)
```

Introduction to Perl: Eighth Lecture

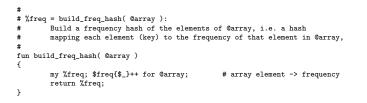
January 2015 10 / 24

## Data Structures On Demand Scaling Agile Data Structures Up

• Plus a bonus function (and a test case, giving **eg10**):



- In reality, there'd be many more such functions, some building and using *xindexlist* instead of, or as well as, *xfreq*.
- Although there's nothing wrong with building %freq and friends independently each time we need them, we might wonder whether we should break such code out:



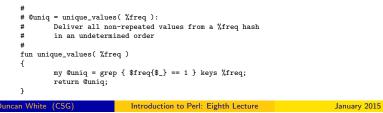
January 2015

## Data Structures On Demand Scaling Agile Data Structures Up

- Now replace that code fragment in other functions with calls: my %freq = build\_freq\_hash(@array );
- Having <code>build\_freq\_hash()</code> available as a separate function opens up the possibility of **prolonging the lifetime** of <code>%freq</code>. Perhaps someone will call both <code>unique\_values()</code> and <code>distinct\_nonunique\_values()</code> with the same array, so why calculate <code>%freq</code> twice?
- Perhaps the caller should do the following:

my %freq = build\_freq\_hash( @array ); my @uniq = unique\_values( \%freq, \@array ); my @nonuniq = distinct\_nonuniq\_values( \%freq, \@array );

- Or, if the order of elements is unimportant, just pass in %freq:
  - my %freq = build\_freq\_hash( @array ); my @uniq = unique\_values( %freq ); my @nonuniq = distinct\_nonuniq\_values( %freq );
- In the latter case, as well as build\_freq\_hash() above, we'd have:



Testing, Benchmarking and Profiling Perl Testing Perl programs

- Perl has several unit testing modules, the simplest is called Test::Simple, but we'll take a quick look at it's big brother Test::More.
- First of all, the basic concept of testing is that you already know what the correct (expected) answer is!
- Test::More has many test functions, we only need three:
  - plan tests => N: How many tests are there in total?
  - use\_ok('module\_name'): Can the given module be successfully loaded?
  - is(\$got, \$expected, \$testdescription): Tests that the string \$got (usually generated from a function you wish to test), is the same as the expected string \$expected, printing out the given test description.
- What shall we test? How about our frequency/unique/distinct values functions, turned into a module frequtils.
- A minimum test might first check that we can load the module:

use Test::More;

plan tests => 2; use\_ok( 'frequtils' ); # how many tests?
# first test.. load module?



13 / 24

- Data Structures On Demand Scaling Agile Data Structures Up
- Plus the remaining functions, rewritten to take %freq:

```
# @nonuniq = distinct_nonunique_values( %freq ):
       Deliver all repeated (non-unique) values from %freq
#
        in an undetermined order
fun distinct_nonunique_values( %freq )
£
        mv %seen:
                                                 # elements we've already seen
       my @nonuniq = grep
                                                # distinct non-unique elements
               { $freq{$_} > 1 && ! $seen{$_}++ } keys %freq;
       return @nonuniq;
}
#
# @distinct = distinct_values( %freq ):
        Deliver all distinct values from %freq
        in an undetermined order
fun distinct_values( %freq )
ſ
        return keys %freq;
l
```

- Adding a test case gives us **eg11**.
- Note the much simpler distinct\_values() implementation now that we don't care about the order also note how we changed the comments for each function to say "in an undetermined order".

```
Duncan White (CSG)
```

Introduction to Perl: Eighth Lecture

```
January 2015 14 / 24
```

## Testing, Benchmarking and Profiling Perl Testing Perl programs

## • Followed by:

```
# my $str = as_string( %hash ):
            Produce a predictable plain text form of a hash
            we've chosen comma separated key:value pairs,
            sorted by key
     fun as_string( %hash )
     Ł
            my @k = sort keys %hash;
            return join( ",", map { "$_:$hash{$_}" } @k );
     3
     my @array = (1,2,1,3);
     my $input = "1.2.1.3":
     my $expected = "1:2,2:1,3:1";
     my %freq = build_freq_hash( @array );
     my $output = as_string( %freq );
     is( $output, $expected.
                                           # second test.. right result?
             "build_freq_hash($input)=$output" );
• This forms eg12. Running it, we get output:
```

- 1..2
  ok 1 use frequtils;
  ok 2 build\_freq\_hash(1,2,1,3)=1:2,2:1,3:1
- Let's check that the test framework is working, by adding
   \$output .= ",6:1" just before the is..

while( (my \$inputarray, my \$expected, @freqtests ) = @freqtests )

is( \$output, \$expected, "build\_freq\_hash(\$input)=\$output" );

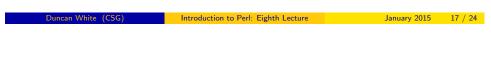
# • As expected, now we get something scarier:

```
1..2
ok 1 - use frequtils;
not ok 2 - build_freq_hash(1,2,1,3)=1:2,2:1,3:1,6,1
# Failed test 'build_freq_hash(1,2,1,3)=1:2,2:1,3:1,6,1'
# at ./eg12 line 36.
# got: '1:2,2:1,3:1,6,1'
# expected: '1:2,2:1,3:1'
# Looks like you failed 1 test of 2.
```

• Scaling this up to more tests of build\_freq\_hash(), we need to

## generalise how tests are represented:

• Need to write new code to run all tests:



Testing, Benchmarking and Profiling Perl Testing Perl programs

• Next, we extend our test framework to extract the 3rd field:

```
#foreach (type,arrayref,expectedstring) triple in @tests
while( (my $type, my $inputarray, my $expected, @tests ) = @tests )
{
    my $input = join( ',', @$inputarray );
    # to be continued
}
```

• Now, we must choose what action to take based on stype. Let's use coderefs and data-driven programming:

• To use this data structure, we carry on in the foreach my \$teststr (@tests) body (from # to be continued):

```
...
# to be continued
my( $testfunc, $funcname ) = @{$testtype{$type}};
my $output = $testfunc->(@array);
is( $output, $expected, "$funcname($input)=$output" );
```

1..7
 ok 1 - use frequtils;
 ok 2 - build\_freq\_hash(1)=1:1
 ok 3 - build\_freq\_hash(2)=2:1
 ok 4 - build\_freq\_hash(1,2)=1:1,2:1
 ok 5 - build\_freq\_hash(1,2,1)=1:2,2:2
 ok 6 - build\_freq\_hash(1,2,1)=1:2,2:2

• Running it, we get output:

• This is simply (eg13):

ſ

3

- ok 7 build\_freq\_hash(1,2,1,3)=1:2,2:1,3:1
- Suppose we wish to generalise further: allow each test to specify

# which function to test, via a 3rd field:

#foreach (arrayref, expected string) in @freqtests

my \$output = as\_string( %freq );

my \$input = join( ',', @\$inputarray );
my %freq = build\_freq\_hash( @\$inputarray );

my @tests = (	<pre># array of (type, arrayref, expectedstr) triples</pre>
"freq", [1],	"1:1", # build_freq_hash() tests
"freq", [1,2,1,3],	"1:2,2:1,3:1",
	"1,2,3", # distinct_values() tests
"uniq", [1,2,1,3],	"2,3", # unique_values() tests
"dnu", [1],	"", # distinct_nonunique_values() tests
"dnu", [1,2,1,2],	"1,2",
);	

```
Testing, Benchmarking and Profiling Perl Testing Perl programs
```

• This only leaves the definitions of the four wrap functions. Here's

Introduction to Perl: Eighth Lecture

wrap\_freq():

```
# $str = wrap_freq( @array ):
# call build_freq_hash( @array ) and then build
# and return a predictable (sorted) representation
# of the result to compare against, as a string
#
fun wrap_freq( @array )
{
    my %freq = build_freq_hash(@array);
    return as_string( %freq );
}
```

- The other 3 are left for you to find in the example tarball.
- This is **eg14** run it, we get output:

```
1..25
ok 1 - use frequtils;
...
ok 4 - build_freq_hash(1,2)=1:1,2:1
...
ok 13 - distinct_values(1,2,1,3)=1,2,3
...
ok 19 - unique_values(1,2,1,3)=2,3
...
ok 23 - distinct_nonunique_values(1,2,1)=1
```

```
Duncan White (CSG
```

}

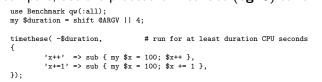
January 2015

18 / 24

- Perl has a module called Benchmark, with a partially OO interface and a procedural interface.
- A Benchmark->new object returns the current time, use it as (eg15):

```
use Benchmark;
my $t0 = Benchmark->new;
                                                # start
# ... put your code here ...
my $x = 100; for(my $i=0; $i<100000000; $i++ ) { $x++; }
my $t1 = Benchmark->new;
                                                # stop
mv $ts = timestr( timediff($t1, $t0) );
print "the code took: $ts\n":
```

• Given several alternative algorithms whose efficiency you want to compare, use the procedural interface (eg16) to run and report:



• There's another example (eg17) using a different benchmark function, \$benchmark\_object = countit( \$time, \$coderef ), to do more flexible benchmarking. Left for you to investigate.

Duncan White (CSG)

Introduction to Perl: Eighth Lecture January 2015

Course Wrapup What haven't we mentioned?

- Perl features such as:
  - typeglobs manipulating symbol tables.
  - Autoloading defining a subroutine AUTOLOAD which handles missing subroutines!
  - Compile time vs run time distinctions, BEGIN and END blocks.
  - Writing Perl code on the fly via eval.
  - Exception handling via eval and die.
  - Perl one-liners (enough times).
- Using the Perl debugger (perldoc perldebug and perldoc perldebtut).
- Perl and graphics building GUIs using Tk or Gtk, visualizing directed graphs via GraphViz and it's friends, constructing image files via ID (useful for CGI programs generating dynamic images).
- Parser generators using Perl especially the awesome vacc-like module Parse::RecDescent.
- Perl threads semaphores, condition variables, mutexes, thread queues etc.
- Interfacing external C libraries into Perl via xs or Inline::C.

January 2015 23 / 24

- Perl's best profiler module is Devel::NYTProf (written by the New York Times). Run one of your Perl programs (eg14 let's say) with: perl -d:NYTProf eg14
- Your program will run 2-3 times slower than usual, then when it finishes, you'll find the nytprof.out file, containing the profiling data.
- Now run a post-processor, nytprofhtml -open. This will produce an HTML report, and open a web browser browsing it. Look at http://www.doc.ic.ac.uk/~dcw/per12014/nytprof/ for an example of a larger Perl program under profiling, you see a table of where time was spent:

0		· ·	0, )		
Calls	Р	F	Exclusive	Inclusiv	e Subroutine
			Time	Time	
14145230	4	1	21.3s	21.3s	NewBoard::cellstatus
5584	3	2	12.7s	27.5s	NewBoard::changeregioncolour
5585	4	3	12.3s	25.1s	NewBoard::extendregion
5528	2	1	4.73s	4.73s	Clone::clone (xsub)
2811488	1	1	4.51s	4.51s	NewBoard::markcellcolour
674063	7	2	1.09s	1.09s	NewBoard::cell

- Then click on any function to see a line by line breakdown of the number of times a line was run, and the time it took.
- Once you know the hotspots, you can consider selectively optimizing them. As in any language, repeated profiling and optimization passes can give dramatic speedups.

January 2015 22 / 24

## Finding More Information about Perl Course Wrapup

- Checkout the Extra Notes document on my website, contains material that didn't fit in the main lectures. New this year: lecture 6's Person/Programmer example done in Moose, a new alternative OO system for Perl.
- O'Reilly's site http://www.perl.com/ (The Perl Resource) is a wonderful source of Perl information. containing links to a multitude of Perl information.
- Our old friend **CPAN**: http://www.cpan.org/.
- The wonderful Perl Journal at http://tpj.com/ which started out as a quarterly paper journal and recently changed to a monthly e-zine in PDF format, still on subscription.
- The Perl Directory at http://www.perl.org/ is a directory of links to other Perl information and news.
- The Perl Monks at http://www.perlmonks.org/ is a forum-based discussion site for all matters Perlish.
- That's all folks! Enjoy your Perl programming and remember the Perl motto: There's More Than One Way To Do It!

Introduction to Perl: Eighth Lecture

• And they're all really good fun!