Sub::Sampling Sub::Space

Brad Bowman <sub@bereft.net>
2012-02-01

Sub::Sampling Sub::Space

Sub::Sampling Sub::Space

These aren't modules, they're just the talk's title.

- Sub::* is an interesting namespace.
- 153 Registered in CPAN, many more unregistered.
- This talk is a tour of some of the curious bits.
- I'll quickly cover some broad themes and then look at some of the more interesting ones in greater detail.

HDTDT

"How do they do that?"

Why Sub:: in Particular?

- My initial scan had a purpose, but there were a lot of diversions to enjoy.
- Manipulating subroutines and code is a higher-order activity.
- This leads to techniques that go beyond ordinary data manipulation.
- Extending the possibilities for code handling extends what coders consider possible.

AUTHORS - Rock-stars and Rising-stars

Damian Conway	Andrew Main (Zefram)
Ricardo SIGNES	Chia-liang Kao (高嘉良)
Curtis "Ovid" Poe	Matthijs van Duin <xmath@cpan.org>
יובל קוג'מן Yuval Kogman	Αριστοτέλης Παγκαλτζής Aristotle Pagaltzis
Florian Ragwitz	David Cantrell
Vincent Pit	Shawn M Moore
Richard Clamp	Johan Lodin
Dan Kogai	MATSUNO★Tokuhiro
David Golden	Steven Haryanto
Simon Cozens	Marcel Gruenauer
Dave Cross	Kang-min Liu "gugod"

Sub:: Makes for Fun Names

My current favourites are:

- Sub::Versive
- Sub::Frequency

But there's so much scope to improve.

[As the author of Devel::file, I clearly enjoy name games]

Modules Themes

- Hooks, Wrappers, Filters, Contracts
- Exports and Imports
- Naming and Package Manipulation
- Timing, Time-limiting, Time-slicing
- Laziness, Memoizing and Retry
- Tweak Subs Properties
- Private Subs
- Param Handling, Signatures
- Current Sub and Tail Recursion
- Pipeline, Chaining, Composing
- Currying, Parameter Binding, Lambda
- Dispatch, Multi-methods
- Sub Information
- Sub Internals

Hooks, Wrappers, Filters, Contracts

- **Sub::Versive** - Subroutine pre- and post-handlers
- **Sub::WrapPackages** - add pre/post-wrappers around subs/packages
- **Sub::Override** - extension for easily overriding subroutines (Mocking)
- **Sub::Prepend** - Prepend code to named subroutines
- **Sub::Assert** - Subroutine pre- and postconditions, etc.
- **Sub::Contract** - Pragmatic contract programming for Perl
- **Sub::Mage** - Override, restore subroutines and add hook modifiers..
- **Sub::Filter** - automatically filter function's return value
- **Sub::Spy** - wrapper that records arguments, return value, exceptions
- **Sub::Uplevel** - apparently run a function in a higher stack frame

Hook::WrapSub, Hook::PrePostCall, Hook::LexWrap, Moose method modifiers.

Exports and Imports - Installation

Sub::Installer - A cleaner way to install (or reinstall) package subroutines

```
  use Sub::Installer;
  $installed_ref = PackageName->install_sub(
                       { subname => $sub_ref });
```

Sub::Install - install subroutines into packages easily

```
Sub::Install::install_sub({
  code => sub { ... }, into => $package, as => $subname });
```

```
This module is (obviously) a reaction to Damian Conway's
Sub::Installer, which does the same thing, but does it
by getting its greasy fingers all over UNIVERSAL.
```

Exports and Imports - Sub::Exporter

Sub::Exporter - a sophisticated exporter for custom-built routines

```
  package From;
  use Sub::Exporter -setup => {
      exports => [ 'plus',
        counter => sub { my $i = 0; sub { $i++ } },
      ],
  };
  sub plus { $_[0] + $_[1] };
  1;
  package To;
  use From 'counter', plus => { -as => 'add' };
  print counter(), ", " for 1..3; print "...\n";
  print "3+5 = ", add(3,5), "\n";
# OUTPUT
0, 1, 2, ...
3+5 = 8
```

Groups, defaults, aliases, prefixes, generated and custom imports... Using it in you module gives your users all these options. Used within Moose.

Exports and Imports - Sub::Exporter::*

- **Sub::Exporter::ForMethods** - helper routines to build methods
- **Sub::Exporter::GlobExporter** - export shared globs with Sub::Exporter collectors
- **Sub::Import** - import routines from most anything using Sub::Exporter
- **Sub::Exporter::Lexical** - to export lexically-available subs with Sub::Exporter
- **Sub::Exporter::Util** - utilities to make Sub::Exporter easier
  `curry_method, curry_chain, merge_col, mixin_installer, like`
- **Sub::Exporter::Simple** - just export some subs
  [This is free software, licensed under: DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE, Version 2, December 2004]

Naming and Package Manipulation

- **Sub::Alias** - alias get_name ⇒ *name*; # no globs
- **Sub::Regex** - Creating Synonymous Subroutines
- **Symbol::Approx::Sub** - calling subroutines by approximate names!
- **Sub::Auto** - declare AUTLOADed subs, with can and inheritance
- **Sub::Delete** - delete subroutines (Sub::Tract...)
- **Sub::Clean** - Clean subroutines with an attribute

```
# Sub::Regex
sub /look(s|ing)?_for/ ($){ ... }
# Sub::Auto
autosub /^get_(\w+)$/ { say "Getting $_[0]..."; }
```

namespace::clean and **namespace::autoclean** are other favourites

[HDTDT: Sub::Regex - Filter::Simple and AUTOLOAD, Sub::Auto - Class::AutoloadCAN Devel::Declare Scope::Guard]

Timing, Time-limiting, Time-slicing

- **Sub::Timekeeper** - calls a function with a stopwatch
- **Sub::Slice** - split long-running tasks into manageable chunks
- **Sub::Timebound** - timebound computations - timeboundretry
- **Sub::Throttle** - Throttle load of perl functions by calling sleep
- **Sub::Fork** - run subroutines in forked process
- **subs::parallel** - enables subs to seamlessly run in parallel
- **Sub::ScopeFinalizer** - execute code on exiting scope
  [HDTDT B::EndOfScope compile time Scope::Guard]

Laziness, Memoizing and Retry

- **Sub::Become** - Syntactic sugar to allow a sub to replace itself
- **Sub::StopCalls** - stop sub calls (make it a constant)
- **Sub::Defer** - defer generation of subroutines until they are first called
- **Sub::Attempts** - alter subroutines to try again on exceptions
- **Sub::Retry** - retry $n times
- **Sub::Frequency** - Run code blocks according to a given probability

Tweak Subs Properties

- **Sub::Prototype** - set subs prototype:
  `set_prototype($code, "&@")`
- **Sub::Attribute** - reliable subroutine attribute handlers
- **Sub::StrictDecl** - detect undeclared subroutines in compilation
- **subs** - pragma to predeclare sub names
- **subs::auto** - Read barewords as subroutine names.
  [HDTDT: B; B::Keywords; Variable::Magic]
- **Sub::Caller** - Add caller information to the end of @_.
- **Sub::Called** - get information about how the subroutine is called
  `already_called, not_called, called_with_ampersand`
- **Sub::Quotelike** - Allow to define quotelike functions (q qq)

Private Subs

Sub::Lexical - implements lexically scoped subroutines

```
my sub this { .. }
```

[HDTDT: Filter::Simple]

Sub::Private - Truly private subroutines and methods

```
use Sub::Private;
sub foo :Private {
    return 42;
}
```

Also **Lexical::Sub** from Zefram, with seems more advanced.

[HDTDT: Attribute::Handlers, namespace::clean, B::Hooks::EndOfScope]

Param Handling, Signatures

- **Sub::Signatures** - proper signatures for subs, including dispatching
  [HDTDT: Filter::Simple]
- **Sub::Parameters** - attributes to unpack params declaratively
- **Sub::ArgShortcut** - simplify functions that use default arguments
- **Sub::NamedParams** - use named arguments with any sub
- **Sub::MicroSig** - microsigs for microvalidation of sub arguments
  `:Sig() Params::Validate::Micro`
- **Sub::Methodical** - call methods as functions (and auto grab $self)
  [HDTDT: B PadWalker Sub::Install Sub::Exporter]

Current Sub and Tail Recursion

- **Sub::Current** - access current code ref with `ROUTINE->()`
- **Sub::Recursive** - Anonymous memory leak free recursive subroutines
  `recursive { $REC->() }`
- **Sub::Call::Recur** - Self recursive tail call invocation (recur of Clojure)
- **Sub::Call::Tail** - Tail calls for subroutines and methods (tail like goto)
  [HDTDT: XS and B::Hooks::OP::Check::EntersubForCV]

Pipeline, Chaining, Composing

- **Sub::Pipeline** - subs composed of sequential pieces
- **Sub::Chain** - Chain subs together and call in succession

> ```
> I then found and considered Sub::Pipeline but
> needed to be able to use the same named subroutine
> with different arguments in a single chain,
> so it seemed easier to me to stick with the code
> ```

- **Sub::Pipe** - chain subs with | (pipe)
- **Sub::Composable** - << composing syntax
  [HDTDT: `use overload '<<' => \&compose;`]
- **Sub::Compose** - like chain but without call overhead

Higher-Order Perl has lazy chains.

… Composing - Sub::Compose HDTDT

Sub::Compose

METHODOLOGY

Currently, this uses Data::Dump::Streamer to deparse the subroutines along with their lexical environments and then intelligently concatenates the output to form a single subroutine. As such, it has all issues that DDS has in terms of parsing coderefs. Please refer to that documentation for more details.

I am working on revamping this so that I manipulate the opcodes directly vs. deparsing. This should have increased performance and, hopefully, will reduce the likelihood of any edge cases. As this is my first foray into the world of perlguts, we'll see how it goes. :-)

Currying, Parameter Binding, Lambda

- **Sub::Curry** - Create curried subroutines (seems complex, with concepts like blackholes and antispices)
- **Sub::Curried** - automatically curried subroutines
- **Sub::DeferredPartial** - Deferred evaluation / partial application. (overloads operators to defer also, nice debug stringify)
- **Sub::Lambda** - syntactic sugar for lambdas in Perl (fn ap)
- **Sub::Lambda::Filter** - experimental source filtering to compile lambdas

```
*flip  = fn f => fn a => fn b => ap qw{f b a};
my $Y  = fn m => ap(
          (fn f => ap m => fn a => ap f => f => a => ()) =>
          (fn f => ap m => fn a => ap f => f => a => ())
        );
(\a -> \b -> { $a + $b }); # S::L::Filter
perl -Mutf8 -C -E 'sub λ { say "in λ" }; λ()'
in λ
```

Also out there **Perl6::Currying**, **AutoCurry**, **Attribute::Curried**

Dispatch, Multi-methods

- **Sub::Multi** - Data::Bind-based multi-sub dispatch (Perl 6 ish) **Data::Bind** - Bind and alias variables
- **Sub::SmartMatch** - Use smart matching to define multi subs
- **Sub::PatMat** - call a version of subroutine depending on its arguments more dynamic based on : when(cond)

  [HDTDT: B, B::Utils walkoptree_filtered opgrep]
- **Sub::PatternMatching** - functional languages' pattern matching
- **Sub::Context** - dispatch subroutines based on their calling context (wantarry dispatch)
- **Sub::Go** - DWIM subs for smart matching: `[1,2] ~~ go { say $_ }`

Sub Information - Sub::Name

Sub::Name

`subname` - (re)name a sub Generally used to give `__ANON__` subs meaningful names.

Sub::Identify

Retrieve names of code references (`sub_name`..)

[HDTDT: B for introspection]

Sub::Name initially for debugging, now common in code generators such as Moose.

`mro` uses `subname` information during method dispatch.

Sub Information - Sub::Information

Sub::Information

single interface to subroutine information

```
use Sub::Information as => 'inspect';

my $code_info = inspect(\&inspect);
print "$_ = ", $code_info->$_, "\n"
  for qw(name package address blessed file
         variables code dump);
```

- Scalar::Util
- Sub::Identify
- Data::Dump::Streamer
- B, B::Deparse
- PadWalker
- Devel::Peek

… - Sub::Information Output

```
name = inspect
package = Sub::Information
address = 20461256
blessed =
file = /usr/local/share/perl/5.10.1/Sub/Information.pm
variables = HASH(0x1416628)
code = $CODE1=sub {
  package Sub::Information; use warnings; use strict 'refs';
  unless ('CODE' eq &Scalar::Util::reftype($_[0])) {
    'Sub::Information'->_croak('Argument to
Sub::Information::inspect() must be a code ref'); }
  return 'Sub::Information'->new(shift @_); };
dump = SV = RV(0x1323d98) at 0x1323d88
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x13836c8
  SV = PVCV(0x1401100) at 0x13836c8
    REFCNT = 3
```

Sub Information - Sub::Nary

Sub::Nary

Try to count how many elements a subroutine can return in list context.

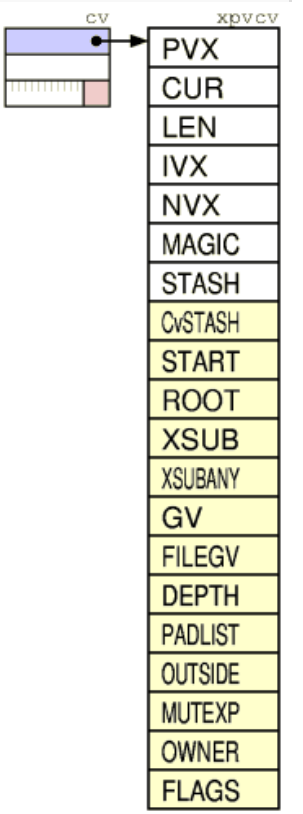
```
use Sub::Nary; use JSON::XS;

sub x {
  if ($ENV{one})      { return 1;      }
  elsif (rand()<0.5)  { return (1,2); }
  else                { return @ARGV; }
}
print encode_json(
          Sub::Nary->new->nary(\&x) );
```

```
{"1":0.5,"2":0.25,"list":0.25}
```

[HDTDT: voodoo]

Sub Internals

```
struct xpvcv {
  HV*           xmg_stash;
  union _xmgu   xmg_u;
  STRLEN        xpv_cur;
  STRLEN        xpv_len;
  HV *          xcv_stash;
  union { OP *   xcv_start;
          ANY    xcv_xsubany;
  }             xcv_start_u;
  union { OP *   xcv_root;
          void   (*xcv_xsub)  (pTHX_ CV*);
  }             xcv_root_u;
  GV *          xcv_gv;
  char *        xcv_file;
  AV *          xcv_padlist;
  CV *          xcv_outside;
  U32           xcv_outside_seq;
  cv_flags_t    xcv_flags
  I32 xcv_depth; };
```



Sub Internals - Sub::Mutate

Sub::Mutate - examination and modification of subroutines

```
    use Sub::Mutate qw(
            sub_body_type
            sub_closure_role
            sub_is_lvalue
            sub_is_constant
            sub_is_method
            mutate_sub_is_method
            sub_is_debuggable
            mutate_sub_is_debuggable
            sub_prototype
            mutate_sub_prototype
    );
```

This module contains functions that examine and modify many aspects of subroutines in Perl. It is intended to help in the implementation of attribute handlers, and for other such special effects.

Sub Internals - Sub::Clone

Sub::Clone

Clone subroutine refs for GC/blessing purposes

A surprising fact about Perl is that anonymous subroutines that do not close over variables are actually shared, and do not garbage collect until global destruction:

```
  use Sub::Clone;
  sub get_callback { return sub { "hi!" }; }
  my $first = get_callback(); my $second = get_callback();
  my $third = clone_sub $second;
  print "# $first == $second\n"; # same refaddr
  print "# $third != $second\n"; # diff refaddr
# CODE(0x7bccc8) == CODE(0x7bccc8)
# CODE(0x790d48) != CODE(0x7bccc8)
```

- Pure Perl - HDTDT: sub { goto &$original }
- XS - cv_clone, real clone for refcounting, sub_name, variables..

Sub Internals - Sub::Op

Sub::Op

Install subroutines as Perl opcodes.
[HDTDT: even greater voodoo]

DESCRIPTION

This module provides a C and Perl API for replacing subroutine calls by custom opcodes. This has two main advantages :

- it gets rid of the overhead of a normal subroutine call ;
- there's no symbol table entry defined for the subroutine.

Subroutine calls with and without parenthesis are handled. [..]

When B and B::Deparse are loaded, they get automatically monkey-patched so that introspecting modules like B::Concise and B::Deparse still produce a valid output.

Unused Sub::Names

Sub::Sequence	Sub::Routine	Sub::Ject
Sub::Standard	Sub::Script	Sub::Class
Sub::PAR	Sub::Scribe	Sub::Mit
Sub::Stance	Sub::Directory	Sub::Mission
Sub::Traction	Sub::Title	Sub::Lime
Sub::Stitute	Sub::Way	Sub::Liminal
Sub::Section	Sub::Marine	Sub::Junctive
Sub::Slice	Sub::String	Sub::Side
Sub::Query	Sub::Born	Sub::Commandant

End

Brad Bowman

Source: https://github.com/bowman/subspace-talk
Presentation: http://bowman.github.com/subspace-talk

Tiny URL: http://tinyurl.com/subspace-talk

- http://search.cpan.org/search?mode=module&query=Sub
- http://cpansearch.perl.org/src/GAAS/illguts-0.09/index.html#cv
- http://cpansearch.perl.org/src/GAAS/illguts-0.09/op.html