Periódico de la Comunidad
Perl de Capital Federal

http://cafe.pm.org/boletines/

Cafe Perl v0.7b

Table of Contents
Cafe Perl v0.7b

Bits and Pieces - Scalars (Mordiditas de aquí y de allá)

Que buena idea el poder escribir sobre algo que ya está escrito y bien depurado (como Perl), pero que a su vez está en pleno período de ebullición (como Perl 6) y que además ya tiene una guía como "Programming Perl". En resumen : o es muuuuy aburrido (porque está todo dicho) o hay que ser muy brillante para poder encontrar esas cositas que andan por ahí perdidas y puedan adquirir luz propia (que no es mi caso). En este caso simplemente se trata de poder entender mejor lo que hay, tratando de ver las mejores prácticas y elementos más aprovechables de Perl 5 y ayudarnos a entender algunas cosas de Perl 6, o por lo menos para que cuando nos tengamos que meter de lleno no se nos haga la noche de repente.

Y para comenzar que mejor que hacerlo con con el capítulo 2 de "Programming Perl" : Bits and Pieces !!!

Los primeros pasos (Atoms, Molecules y Built-in data types) nos ayudan mas bien a comprender ese nivel subyacente en Perl, y son mas parte de la filosofía (qu tratamos en el "Mordidtas ..." del mes anterior), y los pasos siguientes (Variables y Names) nos adentran ya un poco más en la parte teórica, pero sin dejarnos una sustancia como para empezar a embebernos y tener algo de práctica. Para mi una de las vedettes de Perl son los scalars, una especie de contenedor que todo lo abarca y que nos ayuda a despreocuparnos de cómo tratar, donde ponerlo ni cómo almacenarlo : en definitiva algo así como una prenda "wash and wear" (no se preocupe de detalles, sólo úselo !!).

Sencillamente un scalar puede almacenar un número, un string, una referencia u otro dato (que puede ser un scalar o cualquier otro tipo de estructura). Como caso particular puede no almacenar dato alguno, con lo cual se haya solamente definida su estructura. Toda esta magia tiene sus efectos, no es gratis (en Perl 5 simplemente existe y es "la" forma de almacenar un dato) y todo lo que no hagamos nosotros lo hace la CPU, que no es nada más que un recurso que debemos saber usar.

Empecemos por ver cómo internamente es el ciclo de vida de un scalar. Para esto vamos a usar el texto perlguts (Introduction to the Perl API) y a complementarlo con "PerlGuts Illustrated" (http://gisle.aas.no/perl/illguts/).
Para comprenderlo mejor vamos a usar el siguiente script que lo que hace es crear un scalar, asignarle distintos valores a lo largo del mismo y ver su estructura interna :

```
#!/usr/bin/perl

use warnings;
use strict;
use Devel::Peek 'Dump';

my $var1;
print "-" x 10 . " Cuando está vacía no ocupa lugar (NULL) no hay memoria alocada
Dump $var1;
print "\n";

print "-" x 10 . " Lo pongo en contexto NUMBER (IV)\n";
$var1 = 10;
Dump $var1;
print "\n";

print "-" x 10 . " La fuerzo a ser interpretado como STRING (PV): " . $var1 . "\n
Dump $var1;
print "\n";

print "-" x 10 . " Nuevamente como INTEGER (IV) \n";
$var1 += 99;
Dump $var1;
print "\n";
```

```
print "-" x 10 . " Ahora como referencia (RV) \n";
my $var2;
$var1 = \$var2;
Dump $var1;
print "\n";
```

Básicamente cuando creamos un scalar sin asignarle ningún valor se genera una estructura básica, simple y común a todos los tipos de datos que este puede albergar :

```
SV = NULL(0x0) at 0x276cfc
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY)
```

Puede verse que se generó un scalar (SV) que está vacío NULL(0x0) y fue marcado como para usar dentro de un "my" (PADBUSY,PADMY).
Si ahora ponemos un valor entero (10 en nuestro caso), la estructura es la siguiente :

```
SV = IV(0x1930454) at 0x276cfc
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,IOK,pIOK)
    IV = 10
```

Ahora el cambio es mínimo porque se agrega solamente el valor entero (IV = 10) y los flags que indican el cambio a entero (IOK,pIOK). si en lugar de un entero (IV) usáramos un valor con decimales (NV) el caso sería similar.
Pero vayamos un paso mas adelante y forcemos la interpretación de $var1 como un string (PV) :

```
SV = PVIV(0x277dbc) at 0x276cfc
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,IOK,POK,pIOK,pPOK)
    IV = 10
    PV = 0x27f43c "10"\0
    CUR = 2
    LEN = 3
```

Lo notorio de este caso es que se transforma no en un string sino en un número que puede ser usado como string (PVIV) agregándose tres campos más : la representación como string (PV = 0x27f43c "10"\0), su longitud total (LEN = 3) y la longitud real o usada (CUR = 2). También aparecen nuevos flags indicando que es válido como string (POK y pPOK).
Para ser molestos, como de costumbre, vamos a forzarlo nuevamente en el entorno entero (sumándole el valor 99) y ver que pasa :

```
SV = PVIV(0x277dbc) at 0x276cfc
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,IOK,pIOK)
    IV = 109
    PV = 0x27f43c "10"\0
    CUR = 2
    LEN = 3
```

Por suerte ya han dejado de aparecer nuevos campos, pero si prestamos atención se ve que ya no se indica que el valor string es válido (desaparecen los flags POK y pPOK) y el valor de este scalar como string (PV) ya no coincide con el valor entero (IV).
Ni que contar que cuando se le asigna una referencia ya perdemos totalmente el manejo de lo que podría ser una persona en su sano juicio :

```
SV = PVIV(0x277dbc) at 0x276cfc
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,ROK)
  IV = 109
  RV = 0x1978320
  SV = NULL(0x0) at 0x1978320
    REFCNT = 2
    FLAGS = (PADBUSY,PADMY)
  PV = 0x1978320 ""
  CUR = 0
  LEN = 0
```

Particularmente han desaparecido todos los flags relativos a valores IV o PV, aparece el flag indicando que es una referencia (RV), pero curiosamente el valor entero IV sigue existiendo (IV = 109) y también su representación string (PV = 0x1978320 "") pero aparentemente vacío (CUR = 0 y LEN = 0).

Como conclusión esto presenta una dualidad que, además de ser práctica porque nos evita el tener que declarar distintos tipos de variables, va a ser necesario mantenerla y ser manejada por la implementación de Perl, lo que requiere líneas de código que aumentan la complejidad y disminuyen la performance. Si bien vemos que se intenta hacer un uso bastante racional de los recursos (por ejemplo, cuando se pasa de string nuevamente e integer no se actualiza el valor PV sino que se deja desactualizado, y solo lo estará la próxima vez que sea evaluado en contexto string), esto tiene su impacto.

Si les interesa analizar como se hace el upgrade de un tipo de datos a otro, se pueden acercar a la distribución de Perl en forma de source, buscar el archivo sv.c, allí dentro la función Perl_sv_upgrade()) y junto con "PerlGuts Illustrated" (http://gisle.aas.no/perl/illguts/) van a poder seguirlo de cerca.

Volviendo al impacto en performance, vamos a volver a nuestro amigo Benchmark, a tratar de visualizar si existe realmente esta diferencia de performance, si es detectable y tratar de cuantificarla. En principio vamos a trabajar con dos arrays de 1000 elementos (uno con números y el otro con números convertidos s string), los que simplemente vamos a recorrer linealmente e incrementar cada uno de sus elementos :

```
#!/usr/bin/perl

use warnings;
use strict;
use Benchmark qw(:all) ;

my $secs = shift || 1; ## segundos de CPU que dura el test
my $n = 1000; ## cant. de elementos de cada array

##
## La idea de este script es mostrar la diferencia de performance
## entre un numero puro (IV) y un numero en contexto string (PVIV)
##

my $varString = 99; # defino el scalar y almaceno un numero (IV)
$varString .= ""; # ... pero ahora lo convierto a string (PVIV) forzandolo
# a ser interpretado en un contexto string

my $varNum = 99; # scalar sólo numerico (IV)

## ... y después corro el test

my @arrayIV;
my @arrayPVIV;

for my $i (0..($n-1)) {
    $arrayPVIV[$i] = $varString;
```

```
    $arrayIV[$i] = $varNum;
};

my $result = timethese( (-$secs), {
    IV => sub{
            for my $i (0..($n-1)) {
                $arrayIV[$i]++;
            };
        },
    PVIV => sub{
            for my $i (0..($n-1)) {
                $arrayPVIV[$i]++;
            };
        },
    } );

cmpthese( $result );
```

Ejecutando este script con 5 segundos de CPU dio una ventaja de performance entre el 10% y 30% usando el array que posee valores numéricos (IV) versus el que posee valores numéricos pero almacenados también como strings (PVIV). Si ahora hacemos una prueba similar pero haciendo un sort sobre un array de 1000 generados al azar :

```
#!/usr/bin/perl

use warnings;
use strict;
use Benchmark qw(:all) ;

my $secs = shift || 1;
my $n = 1000; ## elementos del array

##
## La idea de este script es mostrar la diferencia de performance
## entre un numero puro (IV) y un numero en contexto string (PVIV)
##

my @arrayIV;
my @arrayPVIV;

getArrayRandom( \@arrayIV, $n );
for my $i (0..$n-1) {
    $arrayPVIV[$i] = $arrayIV[$i]; ## fuerzo a cada elemento a ser número (IV)
    $arrayPVIV[$i] .= ""; ## y a convertirse en string (PVIV)
};

## ... y después corro el test

my $result = timethese( (-$secs), {
    IV => sub{    sort @arrayIV    },
    PVIV => sub{    sort @arrayPVIV;    },
    } );

cmpthese( $result );

sub getArrayRandom($$) {
    my $ref = shift;
    my $items = shift;
```

```
    for my $i (0..($items-1)) {
        my $len = int( rand(6) ) + 1; ## entero de hasta 5 dígitos
        my $num = int( rand( 1 ) * 10**$len );
        push @$ref, $num;
    };
};
```

En este caso la diferencias de performance fueron entre el 30% y 50% a favor del array que posee números convertidos a strings (PVIV) debido a que la comparación default de sort() es string.

Se repite con esto el viejo dicho "lo que no te mata te fortalece", porque la ventaja no ser un lenguaje tipado (no poseer tipos de datos como C, C++ o Java) en cierta forma mata a Perl. Pero bien no todo es malo, porque para el próximo release (Perl 6) se está haciendo que puedan definirse las variables como del tipo que se va a usar, lo cual permite evitar este tipo de penalizaciones. Un ejemplo simple en Perl 6 :

```
#!/usr/bin/pugs

use v6;

my int $var1 = 10;
$var1.say();
```

En principio defino que voy a usar pugs como intérprete (#!/usr/bin/pugs, primer implementación de Perl 6) y que va a ser Perl 6 (use v6). Después defino que hay una variable llamada $var1 que va a ser integer y que es inicializada con el valor 10 (my int $var1 = 10) ... y de paso te recomiendo que si querés empezar a embeberte ya de Perl 6 podés usar Pugs (http://www.pugcode.org).

Con esto se reduce el problema de definir y reusar la variable $var1 como un string, pero realmente "Perl seguirá siendo Perl" y por lo tanto va a seguir permitiendo que se almacenen strings, con lo cual puedo seguir haciendo :

```
$var1 = "Hola";
```

Entonces, dónde está la ganancia. Claramente, y en un primer punto, en la implementación. En particular si hacemos este manejo Perl 6 se va a comportar distinto que Perl 5, no haciendo cache de este valor, y convirtiendo de integer a string cada vez que lo requiera y almacenando sólo el valor integer.

Lamentablemente aún no está la implementación definitiva de Perl 6 (sobre Parrot, la máquina virtual) con lo cual al momento esto es sólo una promesa (digamos que puede estar implementado en Pugs, pero no me atrevería a tomar como una implementación final con la cual hablar sobre performance).

Infografía

- - CPAN (http://www.cpan.org)
- - Módulo Benchmark (http://search.cpan.org/~nwclark/perl- 5.8.7/lib/Benchmark.pm)
- - Módulo Devel::Peek (http://search.cpan.org/~ilyaz/Devel-Peek-0.96/Peek.pm)
- - Pugs (http://www.pugscode.org)
- - Apocalipsis 2 : Bits and Pieces (http://dev.perl.org/perl6/doc/design/apo/A02.html)
- - Sinopsis 2 : Bits and Pieces (http://dev.perl.org/perl6/doc/design/syn/S02.html)
- - Exégesis 2 : Bits and Pieces (http://dev.perl.org/perl6/doc/design/exe/E02.html)
- - Perlguts : introduction to the Perl API (http://search.cpan.org/author/NWCLARK/perl-5.8.7/pod/perlguts.pod)
- - PerlGuts Illustrated (http://gisle.aas.no/perl/illguts/)