Topaz: Perl For The 22nd Century

Chip Salzenberg, VA Linux Systems
<chip@valinux.com>

Singapore Linux Conference, March 2000
# Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What, Who</td>
<td>4</td>
</tr>
<tr>
<td>How</td>
<td>5</td>
</tr>
<tr>
<td>How</td>
<td>6</td>
</tr>
<tr>
<td>When</td>
<td>7</td>
</tr>
<tr>
<td>Why?!</td>
<td>8</td>
</tr>
<tr>
<td>Language Changes?</td>
<td>9</td>
</tr>
<tr>
<td>Value Proposition</td>
<td>10</td>
</tr>
<tr>
<td>Counting Coup</td>
<td>11</td>
</tr>
</tbody>
</table>
Some Scalars Can Change Unpredictably ........................................... 12
“Final” Scalars Don’t Change ......................................................... 13
Changing Identities .................................................................. 14
Scientific Progress Goes “Boink!” .................................................. 15
What Happened to Array and Hash? ............................................... 16
User-Created Implementations ..................................................... 17
OK, Your Turn .............................................................................. 18
What, Who

♦ What Is Topaz?
  • It’s a project to reimplement all of Perl in C++
  • If it comes to fruition, it will be Perl 6
  • The standard for being called “Perl” is very high, so I don’t want to take that name until we have results

♦ Who’s writing it?
  • Me, mostly, for now
  • Internal interfaces must remain fluid during early construction
  • Cooperation over the net is difficult in these circumstances
How

♦ How is it implemented?
  • ISO C++
  • Not C—too low-level
  • Not Objective C—no destructors, no inline functions
  • Not Ada—GNATS written in Ada, imposing burden on targets
  • Not Eiffel—SmallEiffel forbids loading new classes at run-time
How

♦ Where will it run?
  • Anywhere there’s an ISO C++ compiler
  • GCC 2.96 (néé “EGCS”) is portable and free

♦ Windows will be supported

♦ Visual C++ may not be supported
  • It’s very buggy and lags behind the ISO standard
When

♦ When did it start?
  • I’ve had an itch to redesign Perl for three years
  • Official project start was 1½ years ago

♦ When will it be ready?
  • I expected to have something working within a year—oh, well
  • Committed to run simple Perl programs by Perl Conference 4.0 in August, 2000
Why?!

♦ Primary reason: Maintenance
  • Perl’s guts are, well, complicated
  • It’s hard to maintain Perl 5 without long indoctrination into the mysteries of SVs and the magic of MAGIC
  • Some design decisions have made some bugs hard to eradicate
  • Programmer time could better be spent elsewhere

♦ Secondary reason: New Features!
  • Dynamically loaded implementations of basic types
  • Robust bytecode compilation
  • Microperl (Perl without Configure)
  • Configure written in microperl
Language Changes?

♦ Only When Larry Says So
  • Anything deprecated is fair game for removal
♦ He’s the language designer
♦ I’m just the “how” guy (most of the time)
♦ Memories of “*GLOB{IO}” are fresh
  • On the other hand, foreach my was OK :-)
♦ Recent decision: nested each will work
Value Proposition

♦ Abstract base class for all user-visible data is `Value`
♦ Abstract derived classes are
  • `Scalar`
  • `Aggr`
  • `Code`
  • `IO`
Counting Coup

♦ Value is derived from Counted, which implements reference counting

♦ Smart pointer template CountedPtr<> automatically tracks reference counts

♦ For convenience, there are typedefs for smart pointers to each of the main Value types
  • ScalarPtr
  • AggrPtr
  • CodePtr
  • IOPtr
Some Scalars Can Change Unpredictably

♦ It may seem obvious that any scalar can answer the question, “What type are you?”, but in fact many cannot

♦ Tied and overloaded scalars don’t know what their types are until after they have already fetched it (e.g. by calling FETCH)
  • And the answer can change each time the question is asked

♦ C++ lets us express this difference cleanly, though inheritance
“Final” Scalars Don’t Change

- The class Scalar does not include functions to check its value type—“are you a number”, “are you a reference”, etc.

- Derived class FinalScalar does

- So normal scalars are derived from FinalScalar, while magical ones are not

- One method of Scalar is FinalScalar *final()
  - On a FinalScalar, final() just returns this
  - On magical scalars it does the magic—calls FETCH, etc.

- Derived from FinalScalar is FatScalar which is, finally, concrete (non-abstract)
Changing Identities

- Perl values can sometimes change identity
- For example, after `tie $x, 'Pkg'`, the old identity and behavior `$x` is completely hidden by its new tied behavior
- But then, after `untie $x`, the variable should return to its old behavior
- Fully implementing this behavior requires some unelegant trickery:
  - Removing a C++ object from its location
  - Building a new temporary object in its place
Scientific Progress Goes “Boink!”

♦ Thanks to the magic of *transmogrification*, the various *Values* classes can change into each other
  • … as long as they all fit into a small fixed size (architecture-dependent)

♦ This is actually surprisingly portable

♦ ISO C++ says you can kill an old object and put another in its place:

```cpp
void *base = dynamic_cast<void *>(oldobj);
oldobj->OldClass::~OldClass();
new (base) NewClass();
```

♦ The only non-ISO thing about transmogrification is moving the old object away and then putting it back with `memcpy`
What Happened to **Array** and **Hash**?

♦ Pseudohashes happened to them
♦ A pseudohash is an array that can be treated like a hash
♦ You’re not obligated to declare the array in any special way
♦ So Topaz can’t know which arrays will be treated like hashes someday
♦ So the array-like interface and the hash-like interface are folded into a common abstract base class: `Aggr`
User-Created Implementations

♦ The guts of Topaz don’t know or care if you create new kinds of scalars, aggregates, etc.—they just deal with Scalar*, Aggr*, etc.

♦ You should be able to create a new kind of basic data structure—say, a btree hash with always-sorted keys—in an afternoon (or maybe a weekend :-))

♦ Then you should be able to use it by simply making it a derived class of Aggr and putting it into a dynamically loaded extension

♦ The Perl code to use it might look like this:

```
use BTreeHash;
my %h : BtreeHash;
```
OK, Your Turn

♦ Question Time!