The Aranha Web Application Platform

Daniel Silverstone
dsilvers@digital-scurf.org
IRC: Freenode//Kinnison

What is Aranha?

- Dynamic Web application platform
- Comprising a Lua VM with changes...
- ...and significant support code

Who is involved?

- Daniel Silverstone
- Rob Kendrick
- Rici Lake (indirectly)

The history of Aranha

- LHC
- ISAPI/Lua (proof of concept)
- Aranha 1 (abandoned)
- Aranha 2 (in progress)

LHC?

- The Lua Hypertext Compiler
- Simple template interpolation
- Produced static content from the command line
- Lua 3.2 based, no VM changes

ISAPI/Lua

- LHC's content generator with an ISAPI engine
- Generated content each request
- PostgreSQL binding
- Lua 4.0, no VM changes

Aranha 1

- Further modified LHC generator
- FastCGI for portability to non-Zeus webserver
- Each request had own pre-prepared state
- Lua 5.0 (minor VM changes)

Aranha 1 continued

- Its own module loader
- Modular, libdbi binding among others
- Slightly improved interpolation over ISAPI/Lua
- Clever process model of its own

It worked, so why fix it?

- The “improved” interpolation was error prone
- The codebase was messy
- Primary content generator is now ancient.
- Incompatible with 3rd party modules

Aranha 2

- Ground-up rewrite with Lua 5.1
- Class system provided as standard
- Improved diverter
- Application oriented core with simple page-orientation compatibility layer

Aranha 2 continued

- Module system to support Lua5.1 package protocol
- Supports command-line running
- Table comprehensions and other VM changes from Rici Lake
- Documentation strings

The Aranha diverter

- Based on M4's diverter concept
- Used to accumulate strings for various reasons.
- Can be used for HTML, SQL, any text-based stuff.
- Built into the parser, with a small amount of support code around it

The diverter continued

- divert()
- __verbatimdivertedstring()
- __divertedstring()
- undivert()

- string.addformat()

Diverter syntax

- >>Hello World<<
  >>Hello |name|<<
  >>Hello |name#H|<<

That syntax looks very odd

- Consider the following HTML:

```
<table>
  <tr><th>Name</th><th>Age</th></tr>
  <<for name, age in pairs(people) do>>
  <tr><td>|name#H|</td><td>|format_age(age)#H|</td></tr>
  <<end>>
</table>
```

How is that compiled?

- Aranha always compiles your pages as bytecode
- Thus HTML with embedded code needs transforming somehow.
- This is done by wrapping with the >> and << markers

So this is cleverer than Aranha 1?

- Consider the middle of the table from the example:

```
<tr><td>|name#H|</td><td>|format_age(age)#H|</td></tr>
```

- This is compiled to:

```
__divertedstring("<tr><td>%H</td><td>%H</td>", name,
                 format_age(age))
```

- Thus expressions are dealt with in-place rather than post-hoc.

Aranha documentation strings

- Syntax to allow tables and functions to be documented
- Unobtrusive marker: -=-
- Defined syntax for the strings. Similar to doxygen
- Support code built into Aranha to retrieve and parse docstrings

An example docstring (rev)

```
-=-
Reverse the order of a list.

Reverse the order of numeric portion of \t and
return it as a new table.

@param t(table) numerically indexed table to
                reverse
@return rev(table) numerically indexed table of
                   the values of \t in reverse
-=-
```

Aranha's class system

- Single inheritance model with interfaces, abstract classes and metamethod support.
- Classes have the ability to provide __index etc.
- Entire system is ca. 1200 lines of well commented Lua code

What was changed in the VM?

- Some small syntax changes to make table constructor syntax slightly more loose
- Addition of __methindex for OP_SELF
- Addition of __doc and __setdoc for documentation aggregation

A very simple example class

```
Class "counter" {
  function :Constructor(initial)
    self.value = initial or 0
  end

  function :advance()
    self.value = self.value + 1
    return self.value
  end
}
```

Other syntax changes for Aranha

- Aranha also incorporates various syntax changes provided by others:
  - C/C++ style comments from Dan East
  - Table comprehensions from Rici Lake
  - For-loop augmentations from Rici Lake
  - Satisfaction expressions from Rici Lake

Table comprehensions?

- An example of how Aranha isn't afraid to take good ideas from other places.
- Consider this Python statement:

```
keys = [ key for key in dictionary ]
```

- It'd be nice to be able to do similar in Aranha, so we did...

The anatomy of a comprehension

- Valid only in table constructors
- They start like a for statement
- They have a yield section instead of a code chunk
- Then they end
- E.g.

```
keys = { for key,_ in pairs(dictionary) yield key end }
```

What can you yield?

- Yields come in two forms
- List style yields:

  yield <expr>[, <expr>]*
- Map style yields:

  yield “[” <expr> “]” = <expr>
- Or in fact any valid table field

Limitiations of constructors

- Once you reach the 'yield' keyword you can only yield one or more (fixed number at compile time) elements to the constructor

For loop extensions

- To get around that, we added the following for loop extensions:

  ...when <expr>...

  ...while <expr>...

  ...for <another for loop>...

  ...andfor <another for loop>...

A couple of comprehensions

- Two simple examples:

```
{ for i = 1, #T when i%2 == 0 yield T[i] end }
```

```
{ for k,v in pairs(env)
    when tonumber(v) ~= nil
    yield [k] = v
end }
```

Satisfaction expressions

- A satisfaction is an expression of the form:
  `<varlist> = <exprlist> satisfies <expr>`
- The <varlist> can be used in the new scope and the expression evaluates to the value of <expr>
- Can be used with if/while/when

Very simple satisfaction example

```
if ok, message = some_func() satisfies ok then
    wahey(message)
else
    darn(message)
end
```

Future development plans

- Integration with LuaJIT 1.1.2
- Caches
- Finish 5.1 pure module support
- Standard modules: DBI, MD5 etc
- Fix bugs
- Implement good suggestions made to me today/tomorrow.

Any questions?