Sublanguages
Encoding programming paradigms in Ruby

Christian Neukirchen
Editor in Chief of Anarchaia

http://chneukirchen.org/talks

What is this talk about?

Ruby

C++ Java PHP Visual Basic

It's not about inferior languages!

Instead…

- It's about doing things in ways you didn't expect…
  - …but doing them in Ruby

Overview

- What are sublanguages?
- `sloop`: prototype-oriented programming
- `spawn`: Erlang-style concurrency
- `solve`: logic programming

Sub-what?

- Sublanguages are like embedded Domain-Specific-Languages (“DSLs”)…
- …just not Domain-Specific!

Sublanguages

- Embedded, general purpose "programming languages"
- Solutions for solving general problems
- Can use the full power of Ruby
- Usable for real programs!

Self-made Restrictions

- A lot is possible…
- …but there are some things I'd rather avoid:
  - Modifying core classes
  - Breaking code that used to work
  - Being *completely* inefficient

Ruby-imposed Restrictions

- Speed
  - Not comparable with native implementations
- Syntax “restricted” to Ruby
- Evaluation restricted to Ruby
  - We (still?) have continuations, but I’d like to avoid them because of the self-made restrictions

Three sublanguages: Techniques used

- `sloop`: prototype-oriented programming
  - `method_missing` on steroids
- `spawn`: Erlang-style concurrency
  - Core-class inheritance
  - Making up object identities
- `solve`: logic programming
  - Expression construction by operator overloading
  - Goal-directed evaluation with blocks

sloop

- Abolish the class system!
- Build-your-own method dispatch
  - Example:

```
Account = sloop {
  self.balance = 0

  def_deposit  { |v| self.balance += v }
  def_withdraw { |v| self.balance -= v }

  def_inspect { “#(an account with $#{balance})” }
}
```

Example:

```
my_account = Account.clone
puts my_account.inspect
my_account.deposit 1000
puts my_account.inspect
```

```
#(an account with $0)
#(an account with $1000)
```

How it works

- *Excessive* use of `method_missing`
- If the name matches `/^def_/`
  - Set the slot to a `Sloop::Proc`
- If the name matches `/=$/`
  - Just set the slot

How it works, part 2

- Else…
  - If the slot exists
    - Retrieve it
    - If it's a `Sloop::Proc`, run it
    - Else, return the value
  - Else, try looking in the `_parent` slot

Pros

- Everything is possible
  - A flexible mixin/“inheritance” scheme is included
    - Next stop: conditional traits?
      - “If the balance is bigger than 1,000,000, the object automatically turns into a `RichAccount`”
- Despite the `method_missing`s, pretty safe to use

Cons

- Slower dispatch times (the classic Ruby disease)
- It's totally different compared to the Ruby class system

Use when…

- You need to model complex relationships (mainly business logic)
- You have lots of special-purpose objects (few instances of a lot of classes)
- You want to prepare for your move to Io (Ewww?) or Self (Zzzz?)

spawn

- Erlang-style concurrency for Ruby
- ~~Threads~~Processes send each other messages
- No shared memory between threads
- Easier to program (no locking)
- Scales better

An example:

```
adder = spawn {
  sum = 0
  loop {
    recieve { |sender, msg, *args|
      case msg
      when :add    then sum += args.first
      when :result  then sender.reply sum
      end
    }
  }
}

10.times {
  spawn { |process| adder.send :add, rand(10) }
}
p adder.syncmsg(:result)
```

Implementation

- We inherit `Spawn::Process` from `Thread`
  - …and add a `queue` attribute
  - …and some helpers to read and write that queue
- Only single-process concurrency so far, but should be easy to scale with help of DRb
  - …or even a “proper” message queue

Synchronous messaging

- You can use `syncmsg` to send a message and wait for a *reply* to it
  - Usually done by passing a handle to the current process
- But how can we tell that we really meant *this* message?
- `object_id` of both processes is the same
- …so let's wrap them with a `ProcessWrapper`
  - it only forwards everything, but has an unique `object_id`

Pros

- Easy to use, when you have the appropriate mindset
- No more mutexes
- Helps designing for scalability

Cons

- Only uses Ruby's threads so far
  - Which, albeit "lightweight" still are *huge* in comparison to Erlang's (~40K vs. only 1K)
- …and occasionally flaky
  - Please don't use for emergency telephony services!
  - Look at Ruby's implementation for detail

Use when…

- You are looking for a more "natural" way to do concurrency
- You want to write code that scales easily
- You think Ruby on Rails is a lot cooler than ErlyWeb

solve

- Logic programming for Ruby
- Rudimentary constraint satisfaction
- Example:
  - David is the son of John
  - Jim is the son of David
  - Steve is the son of Jim
  - Nathan is the son of Steve

```
def parent?(a, b)
  ((a == “David”)  & (b == “John”))  |
  ((a == “Jim”)    & (b == “David”)) |
  ((a == “Steve”)  & (b == “Jim”))   |
  ((a == “Nathan”) & (b == “Steve”))
end

def anchestor?(a, b)
  z = Solve::Variable.new   # anonymous variable
  parent?(a, b) |
    parent?(a, z) & Then.do { anchestor?(z, b) }
end

child     = Solve::Variable.new(:child)
anchestor = Solve::Variable.new(:anchestor)

solve(   (child == “Nathan”) &
           anchestor?(child, anchestor)
) { |result| p result }
```

Result:

```
{:child=>"Nathan", :anchestor=>"Steve"}
{:child=>"Nathan", :anchestor=>"Jim",
                                :_1=>"Steve"}
{:child=>"Nathan", :anchestor=>"David",
                        :_1=>"Steve", :_2=>"Jim"}
{:child=>"Nathan", :anchestor=>"John",
        :_1=>"Steve", :_2=>"Jim", :_3=>"David"}
```

Creating predicates from data structures

```
def parent?(a, b)
  Solve.forany({“David”  => “John”,
                “Jim”    => “David”,
                “Steve”  => “Jim”,
                “Nathan” => “Steve”}) do
                |child,     father|
    (a == child) & (b == father)
  end
end
```

```
def Solve.forany(enum, &block)
  enum.inject(Solve::False) { |a,e| a | block[e] }
end
```

HTF does that work?

- First, Desugaring:
  - `a | b` ⇒ `Or.new(a, b)`
  - `a & b` ⇒ `And.new(a, b)`
  - `~a` ⇒ `Not.new(a)`
  - `a == b` ⇒ “Variable with expected value `b`”

Then…

- Solve tries to *unify* the expression
  - A variable unifies if the value is unset
    - Then it sets the expected value to the given one
  - …or if the value matches the expected value
- All values are stored in a dynamically scoped environment that's passed around implicitly

Logical operators

- `And` unifies if all subclauses unify
- `Or` unifies for every subclause that unifies
- `Not` unifies if the subclause doesn't unify
- `True` always unifies
- `False` never unifies

What does *unify* mean?

- In `solve`, unify means "calls a block"
- The whole thing just calls a lot of blocks!
- Attribution for the idea: YieldProlog
  http://yieldprolog.sourceforge.net/
  (which lacks the sugar)

```
class Or
  def unify
    @elts.each { |e|
      e.unify { yield }
    }
  end
end
```

```
class And
  def unify
    @a.unify {
      @b.unify { yield }
    }
  end
end
```

```
class Not
  def unify
    succeed = false
    @expr.unify { succeed = true }
    unless succeed
      yield
    end
  end
end
```

Therefore…

- If we don't yield, the "trial and error" stops
- The final yield calls the block given to `solve` with the current environment
- unify is a kind of visitor for the expression tree

Pros

- Elegant design
- Clever syntax
- Nice pattern
- Extensible (e.g. `digit.oneof 0..9`)

Cons

- Lots of method calls (yawn)
- Totally generic and unoptimized
  - Anyone want to hook a constraint-solver like Gecode into it?
- Recursive queries need to be protected (with `Then.do`)
- Due to unadept precedence you may need lots of parentheses (yay for Lisp)
- A bit difficult to debug

Use when…

- You need logic programming but don't know Prolog or can't embed it
  - (It's non-trivial to use `solve` without some knowledge of logic programming, though.)
- You like debugging recursive programs (a great way to learn ;-))
- The technique is useful for developing all kinds of query languages (cf. Criteria)

Summary

- If your head smokes now, that's alright
- But talking about trivial things would have been a waste of time, no?
- When you're writing a logic program in Ruby, it doesn't really look like Ruby anymore…

Sublanguages!

- Enable *multi-paradigm* programming
- *"A paradigm is a key model, pattern or method (to achieve certain class of goals/objectives)."*
  — Wikipedia
- That means:
  - We can express *foreign paradigms* in Ruby

Prototyped programming

Logic programming

Concurrency

Why?

- Ruby is very powerful…
- …but not too powerful
- That makes the language flexible enough, but also recognizable enough
  - Anti-Example: Lisp
  - We still can leverage the full language
    - That implies we can mix paradigms

Now you can…

write

concurrent

logic programs

that are developed in a

prototyped manner

(please don't!)

Thanks for your attention

- Slides: http://chneukirchen.org/talks
- Code: http://chneukirchen.org/repos/sublanguages

Thanks to: David Föll, Robert Retzbach and Alexander Kellett for reviewing the slides.