Refining refinements

Shugo Maeda

2013-05-30T17:30:00

Network Applied Communication Laboratory Ltd.

What are refinements?

Local class extensions

- Adding methods to, or changing methods in existing classes
- Activated in a local scope

Example

```
module Rationalize
  refine Fixnum do
    def /(other) quo(other) end
  end
end

p 1 / 2 #=> 0
using Rationalize
p 1 / 2 #=> (1/2)
```

Basic concepts

Packaged in a module

- Multiple refinements can be included in a single module

Module#refine(*klass*)

- Defines a new refinement for *klass* in the receiver

main.using(*mod*)

- Activates refinements in *mod*
- main is self at toplevel (i.e., only available at toplevel)
- Refinements are activated only in the file using is called
    - i.e., if control is transferred to another file (e.g., by a method call), refinements are deactivated

What are refinements for?

For better compatibility

- Trying breaking changes
- Backward compatibility

For internal DSL

- Extensions for built-in classes

Trying breaking changes

Breaking changes

- Changes which break other code
  - Incompatible changes in existing methods
  - Method removal
  - Method addition (changes the behavior of method_missing)

Implement such changes as refinements

- They are activated locally
- Files not using them never be broken

Then, make them global if necessary

Example

- Integer#/

Backward compatibility

Save existing code from braking changes

Example

```
module OldChars
  refine String do
    def chars; each_char; end
  end
end
p "foo".chars #=> ["f", "o", "o"]
using OldChars
p "foo".chars #=> #<Enumerator: "foo":each_car>
```

Internal DSL

Extensions for built-in classes

- Literals
- Fixnum, Bignum, Float, String, Symbol, Array, Hash...

Limit extensions to DSL using refinements

Case study

radd_djur

- https://github.com/shugo/radd_djur
- Packrat parser combinator library

Packrat parser

Top-down parser

- with backtracking
- and memoization

Parser combinator

Allows you to combine small parsers into one big parser

Each parser is a monad

What's a monad?

- Ask Haskell guys

PEG

Parsing Expression Grammar

Formal grammar

Ambiguity free

Example of PEG

```
additive  <- multitive '+' additive
           / multitive
multitive <- primary '*' multitive
           / primary
primary   <- '(' additive ')'
           / digits
digits    <- [0-9]+
```

Example of a RaddDjur parser

```
using RaddDjur::DSL
g = RaddDjur::Grammar.new(:additive) {
  # additive <- multitive '+' additive / multitive
  define :additive do
    :multitive.bind {|x|
      "+".bind {
        :additive.bind {|y|
          ret x + y
        }
      }
    } / :multitive
  end
}
```

define

```
define(name, parser)
define(name) { parser }
```

Define a new nonterminal and its rule

bind

>>= (bind) in Haskell

sequence in PEG (e.g., a b)

Example:

```
a.bind { |x| b.bind { |y| ret [x, y] }
```

- First, invoke the parser a
- If succeeded, invoke the parser b on the remainder of the input string left unconsumed by a
- If succeeded, return the result

ret

return in Haskell

Used to return the parsing result

Example:

```
(?0..?9).bind { |x| ret x.to_i }
```

- Return the value of x.to_i as the parsing result of (?0..?9).

/

/ in PEG (e.g., a / b)

Example:

```
a / b
```

- First invoke the parser a
- If succeeded, return a's result
- Otherwise, invoke the parser b
- If succeeded, return b's result

Parser literals

Symbol

- Parser for nonterminal symbols
  - e.g., :additive, :multitive

String

- Parser for terminal symbols
  - e.g., "+", "lambda"

Range

- Parser for characters within the specified range
  - e.g., ?0..?9, ?a..?z

Implementation of parser literals

```
refine Object do
  def bind(&block)
    to_parser.bind(&block)
  end

  def to_parser
    raise TypeError
  end
end
```

Implementation of parser literals (cont'd)

```
refine Symbol do
  def to_parser
    Grammar::Parser.new(&self)
  end
end
refine String do
  def to_parser
    Grammar::Parsers.string(self)
  end
end
```

How does it work?

```
using RaddDjur::DSL


"-".bind { :digits.bind { |x| ret -x.to_i } }
```

```
"-".to_parser.bind {
  :digits.to_parser.bind { |x|
    ret -x.to_i
  }
}
```

Pros/cons of refinements

Pros

- Intuitive notation for DSL (like monkey patching)
- Isolation (unlike monkey patching)

Cons

- High context (unlike monkey patching)

A limitation of refinements

Only file-scoped refinement activation

```
using RaddDjur::DSL
g = RaddDjur::Grammar.new(:additive) {
  ...
}
```

- using is necessary for each file
- Incompatible refinements can't be used in the same file

How to refine refinements

Block-scoped refinement activation

```
g = RaddDjur::Grammar.new(:additive) {
  # RaddDjur::DSL is activated here.
}
x = Foo.new {
  # another incompatible refinement is
  # activated here.
}
```

- How to achieve this?

PROPOSAL using: option of instance_eval

Example

```
def initialize(&block)
  instance_eval(using: RaddDjur::DSL, &block)
end
```

- The value of using: option is a module, or an array of module
- The refinements defined in the module(s) are activated only in the given block

Considerations

Why instance_eval?

- instance_eval is often used for DSL
- self switching and refinement activation should be able to be used at the same time

How about other eval methods?

- I have no idea for eval and module_eval
- But, instance_exec should have the using: option

Readability

- Implicit refinement activation may be confusing for users

Performance issue

- Please help me, SASADA-san

Conclusion

What are refinements for?

- For better compatibility
- For internal DSL

How to refine refinements

- using: option for instance_eval

Thank you!

Any questions?