Ruby Tooling: State of the Art

Tor Norbye and Martin Krauskopf

Sun Microsystems
http://www.sun.com

Session TS-9972

Goal

Demonstrate that full-fledged integrated development environments (IDEs) can significantly boost programmer productivity for dynamic languages such as Ruby.

Ruby Tooling: State of the Art

- Much of the work is not Ruby-specific
  - The techniques apply to most other dynamic languages
- Expect similar tooling elsewhere

Non-Goals

- Coverage of the latest type inference research in academia
  - This talk focuses on state of well-known Ruby tools
  - Rapidly moving target!
- Discuss all aspects of Ruby development: Rails…
- Fair and balanced IDE shoot-out
  - Editors: Textmate, Emacs, Vim
  - Commercial IDEs: IntelliJ, “Ruby In Steel”, Komodo IDE
  - Free IDEs: Eclipse RDT, RadRails/Aptana and DLTK, NetBeans™ IDE

Agenda

Editors vs. IDEs
Debugging
Code Templates
Code Completion and Type Attribution
Goto Declaration
Refactoring
Demo

Agenda

Editors vs. IDEs

Debugging

Code Templates

Code Completion and Type Attribution

Goto Declaration

Refactoring

Demo

“We Don’t Need No Stinking IDEs”

- “You Java™ technology people need IDEs to help you with all that boiler plate code; my language is much cleaner than that”
  - DRY principle—don’t repeat yourself
  - Real programmers don’t use a crutch like an IDE
- I don’t need all that bloat—I have Emacs!
  - **E**ight **M**egabytes **A**nd **C**ontinually **S**wapping

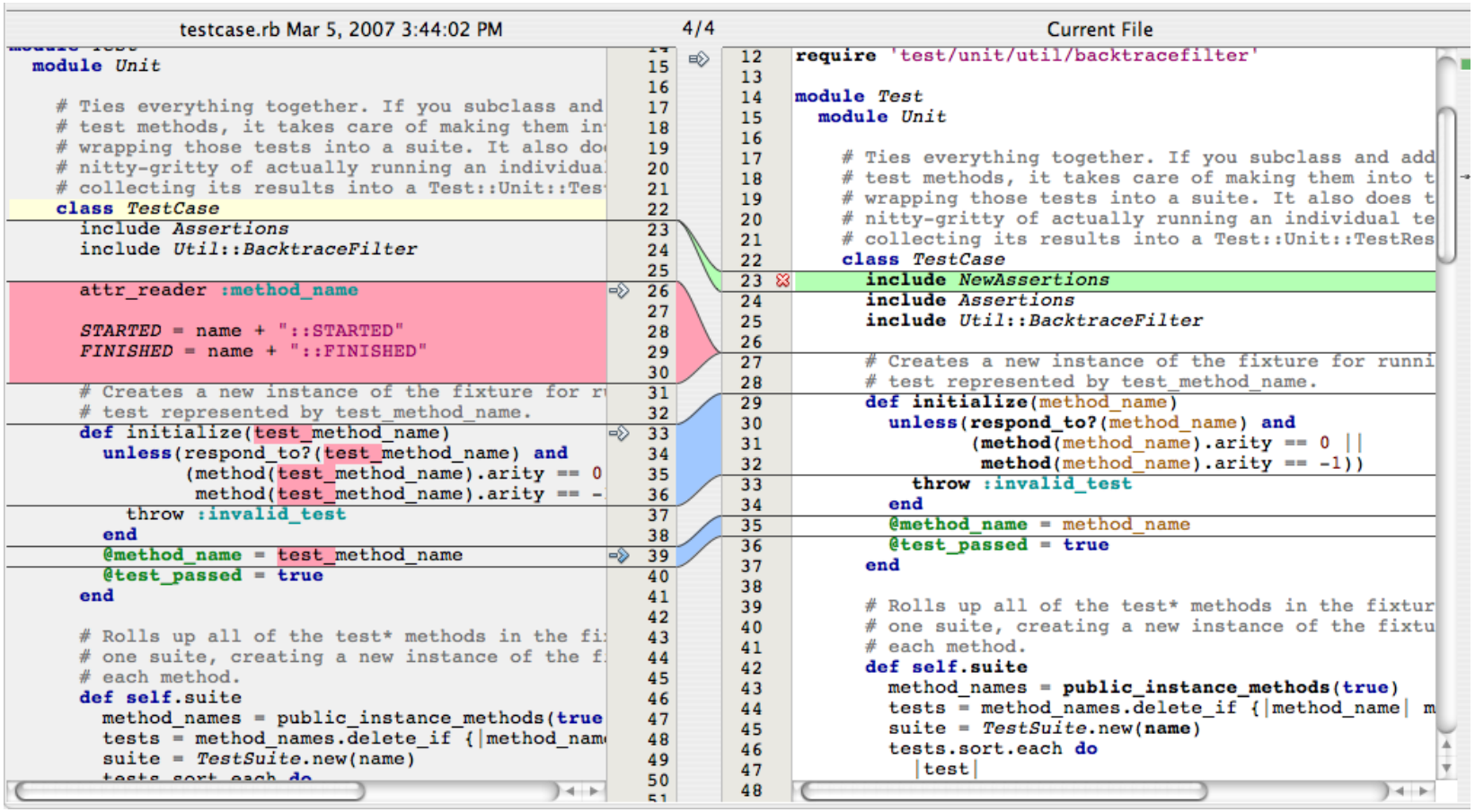
Don’t Forget Vim!

Source: http://eigenclass.org/hiki/rcodetools-screenshots

Editor vs. Full-Blown IDE

- Emacs is an IDE
  - For some, it's a login shell, a mail tool…
- IDE facilities
  - Support for all coding related tasks
  - Team collaboration: tight version control system integration, integrated chat and code sharing
  - Tasks/TODO management, project system
  - Debugging infrastructure: balloon eval, thread view…
  - Plug-in management

Local File Editing History Diff



Agenda

Editors vs. IDEs

Debugging

Code Templates

Code Completion and Type Attribution

Goto Declaration

Refactoring

Demo

Tools Supporting Ruby Debugging

- **CLI tools**
  - debug.rb, ruby-debug-cli gem, breakpointer
- **Power editors**
  - Emacs, Textmate, (Vim?)
- **IDEs**
  - Arachno, DLTK, FreeRIDE, Komodo, Mr. Guid, Mondrian, NetBeans, RadRails, RDT, Ruby In Steel,...

CLI Tools

- debug.rb
  - `ruby -rdebug hello.rb`
- Breakpointer
  - `<rails_app>/script/server`
  - `<rails_app>/script/breakpointer`
  - `breakpoint call`
- ruby-debug-cli gem
  - `rdebug hello.rb`

Tools Supporting Ruby Rebugging

- CLI tools
  - debug.rb, ruby-debug-cli gem, breakpointer
- **Power editors**
  - Emacs, Textmate, (Vim?)
- **IDEs**
  - Arachno, DLTK, FreeRIDE, Komodo, Mr. Guid, Mondrian, NetBeans, RadRails, RDT, Ruby In Steel...

Techniques

- Kernel#set_trace_func(event_handler)
- Kent Sibilev's **`ruby-debug-base`**
  - **`Handler: at_line, at_breakpoint at_catchpoint, at_tracing`**
  - Native C Ruby extension
- Others (hacking interpreter)

Techniques

- **Kernel#set_trace_func(event_handler)**
- Kent Sibilev's **`ruby-debug-base`**
  - **`Handler: at_line, at_breakpoint at_catchpoint, at_tracing`**
  - Native C Ruby extension
- Others (hacking interpreter)

Techniques

- Kernel#set_trace_func(event_handler)
- **Kent Sibilev's `ruby-debug-base`**
  - **`Handler: at_line, at_breakpoint at_catchpoint, at_tracing`**
  - Native C Ruby extension
- Others (hacking interpreter)

Techniques

- Kernel#set_trace_func(event_handler)
- Kent Sibilev's **`ruby-debug-base`**
  - **`Handler: at_line, at_breakpoint at_catchpoint, at_tracing`**
  - Native C Ruby extension
- **Others (hacking interpreter)**

Implementing frontend

- From scratch
- Choose technique
  - ruby-debug-base for C Ruby
  - set_trace_func for JRuby
- Communication protocol
- etc.
- Reinventing the wheel
- Slow progress of Ruby debugging

Solution: Debug-Commons

- Open source rubyforge.org project
  - Common effort
  - RDT, NetBeans
  - Nice contribution from Markus Barchfeld (RDT)
  - DLTK and others?
- IDE-independent
- Language-independent

Future Works, Aims

- debug-commons
  - frontend entry point
  - Debugging standard in the future (as Java's JPDA)
  - A lot of work requires more people to get involved
  - Gain for community as well as for all frontend implementers
- jruby-debug (fast debugging for JRuby)
- Cross-language debugging

Summary

- Ruby debugging quickly becoming mature
  - Full-fledged and **fast** debuggers frontends **are** available
- Cooperation is working well
- Join the debug-commons project
- Still a lot of work to be done

For More Information

- Debug Commons
  - http://debug-commons.rubyforge.org/
  - http://debug-commons.rubyforge.org/misc/ruby-debugging.html
- Ruby-debug
  - http://www.datanoise.com/ruby-debug/
  - http://rubyforge.org/projects/ruby-debug/

Agenda

Editors vs. IDEs

Debugging

Code Templates

Code Completion and Type Attribution

Goto Declaration

Refactoring

Demo

IDEs and Boilerplate

- Java IDE: add a private field `foo` and apply encapsulate field refactoring or quick fix
  - IDE generates getter and setter methods
  - IDE can collate these into a logical property in navigator
- Ruby: add `attr_accessor :foo` and the language/runtime provides getter and setter methods
- However, you still have to code…
  - Lots of common idioms
  - Rails has a large body of popular templates

Code Templates and Snippets

- Live code templates
  - Linked substitution of **related** and **logical** variables
  - With “DRY”, related variables not as common in Ruby
- Semantic information helps:
  - Choose variable names guaranteed to be unique: no accidental aliasing or side effects
  - Templates referring to existing variables can automatically use the right one
    - For example, a template referencing a hash can expand to using a hash present in the local scope

Live Template Example

```
hashkeys:
${name type="Hash"}.each_key { |${o new}| yield(${o}) }

def foo
  o = 50
  new_topic_values = { :title => "AnotherTopic" }
  new_array = [1,2,3]

  # Expand template hashkeys
  new_topic_values.each_key { |p| yield(p) }

end
```

Linked logical variable: o

Code Templates and Snippets

- `${param_name type="Fixnum"}`
  - Requires parameter type to be of a specific type
  - This may help the IDE match an existing variable with this parameter
- `${param_name new}`
  - The variable name will be a new, unique local variable
  - There are similar variables for fields, classes, constants, etc.
- `${param_name default="file"}`
  - Specify a default name to be used

Agenda

Editors vs. IDEs

Debugging

Code Templates

Code Completion and Type Attribution

Goto Declaration

Refactoring

Demo

Static Typing vs. Dynamic Typing

Example 1:

```
public List javaUnmodifiableList(List list) {
    return new Ar<complete>   ==> ArrayList

def ruby_unmodifiableList(list)
    return Ar<complete>       ==> ?
```

Example 2:

```
public void javaTrim(String s) {
    return s.tr<complete>     ==> s.trim()

def ruby_chomp(s) {
    return s.ch<complete>     ==> ?
```

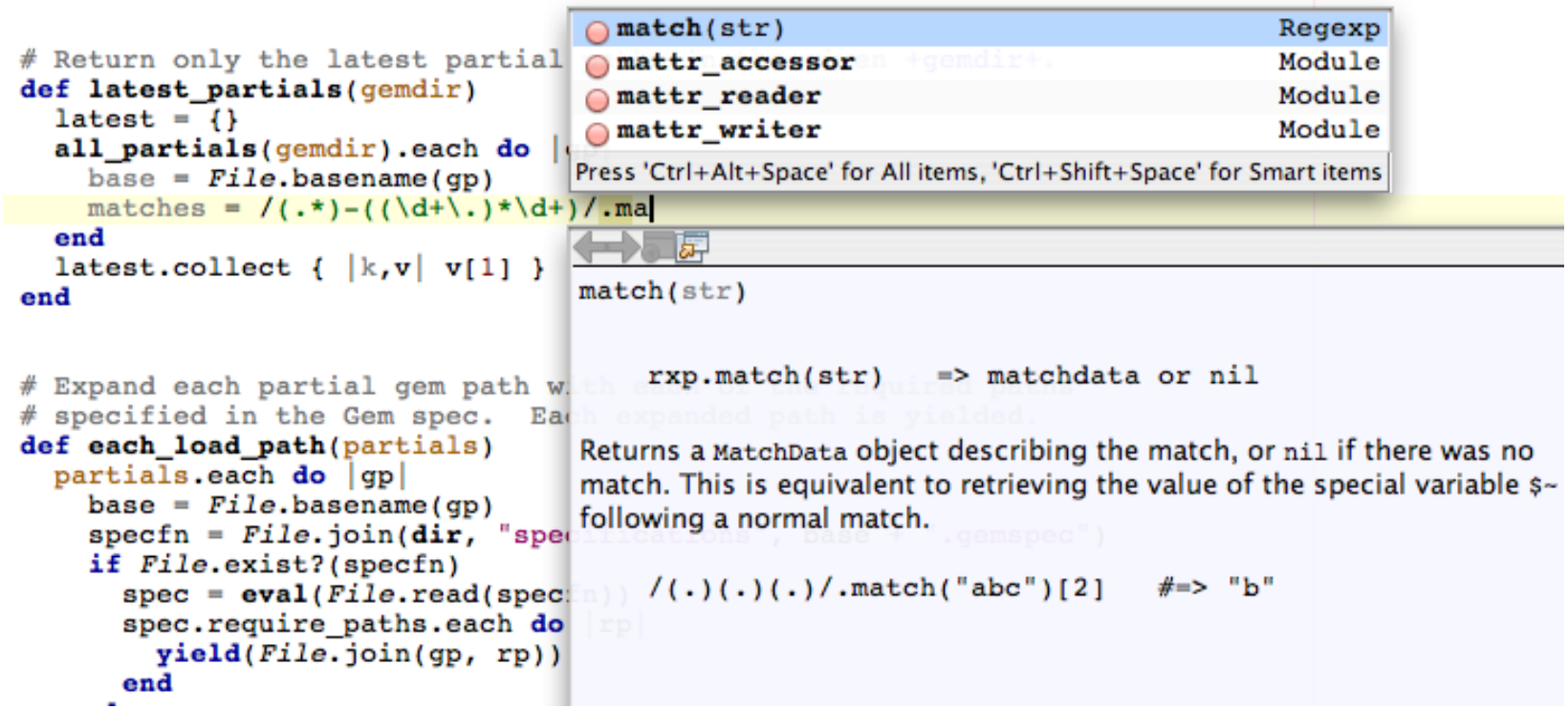
Code Completion

- **Not** the same as automatic { and } pair matching, inserting a matching “end” for a “begin”, etc.
- Goal: Determine the exact set of applicable methods, instance variables, class variables, classes, or modules under the cursor
- Many uses in the IDE—not about keystrokes
  - Completion pop-up, parameter help, documentation pop-up
  - Goto declaration
  - Rename refactoring, other refactorings

Completion Scenarios

- Exploring an unfamiliar class
- Exploring available modules and classes
- Learning or confirming method parameter usage
- Learning the language
  - Completion on %: show and describe %q, %Q, %r,…
  - Completion on $: describe globals - $*, $<,…
  - Completion in \ in regexps: describe \A, \S, \w,…
  - Completion on Ruby keywords: explain yield, unless…

Completion Usage Example



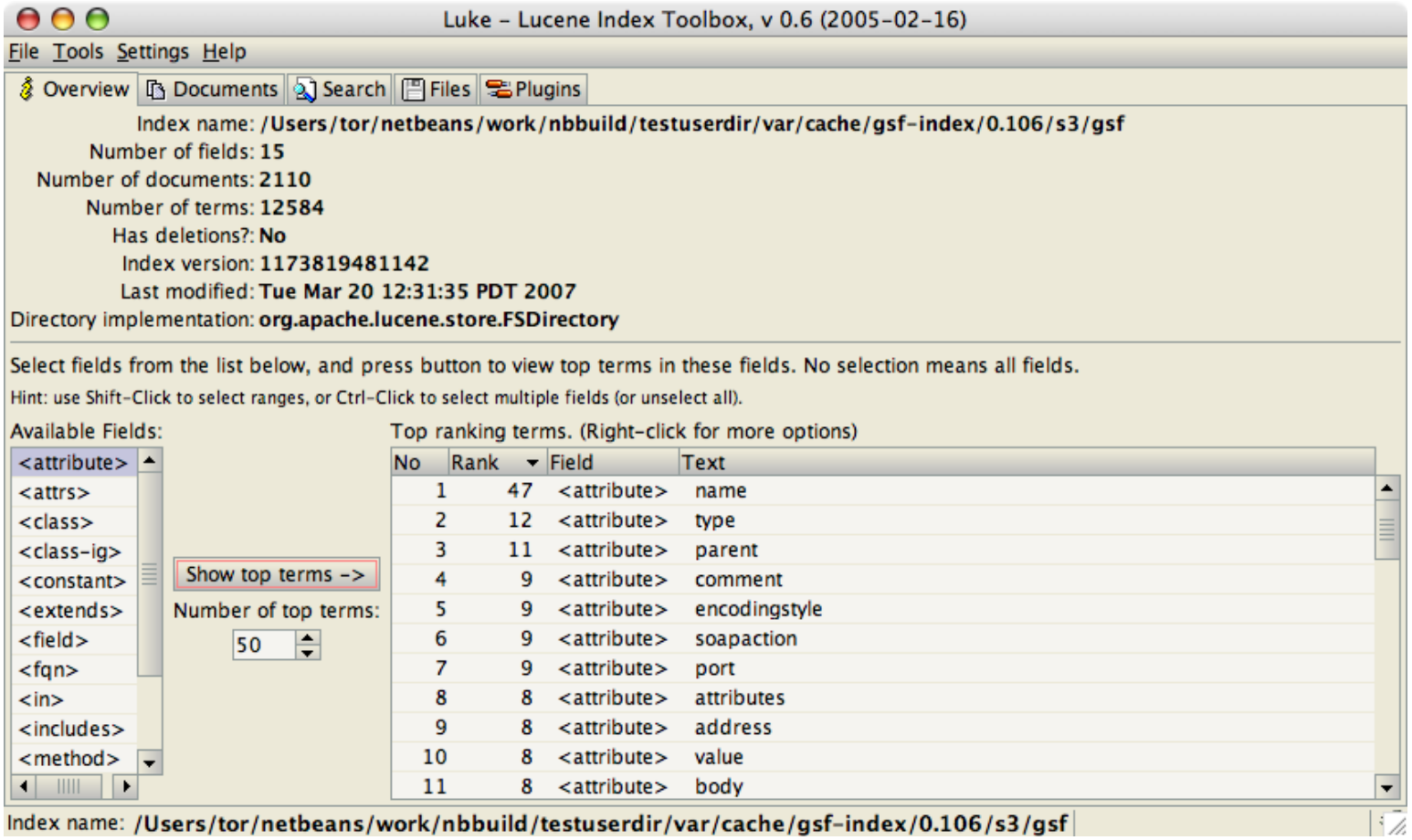

```
# Return only the latest partial
def latest_partials(gemdir)
  latest = {}
  all_partials(gemdir).each do |
    base = File.basename(gp)
    matches = /(.*)-((\d+\.)*\d+)/.ma
  end
  latest.collect { |k,v| v[1] }
end


# Expand each partial gem path w
# specified in the Gem spec.  Ea
def each_load_path(partials)
  partials.each do |gp|
    base = File.basename(gp)
    specfn = File.join(dir, "spe
    if File.exist?(specfn)
      spec = eval(File.read(spec
      spec.require_paths.each do
        yield(File.join(gp, rp))
      end
```

The Index

- Completion prerequisite: global knowledge
  - Know about user classes
  - Know about libraries: Ruby built-in libraries AND gems
  - This is typically what separates editors from IDEs
- Index allows fallback mechanism: prefix matching
- unknown.**`descr`** matches
  - OCI8:**`descr`**ibe(name)
  - Exception::**`descr`**ibe_blame
  - Generators::HtmlMethod::**`descr`**iption

The NetBeans IDE Ruby Lucene Index



Singleton Method Completion

- Singleton method completion
  - File.exi => File.exists, String.ne => String.new
- Algorithm
  - Index knows about singleton methods from parse trees
  - Determine the exact class on the left hand side
  - May involve looking at require statements transitively
  - Add all its methods known to be singleton methods
  - Iterate recursively up the chain to include inherited methods from superclasses and module mixins

Object Literal Completion

- Literal method completion
  - “foo”.gs => String.gsub, 5.ea => Fixnum.each,…
- Completion algorithm
  - Look at lexical tokens, determine corresponding type
  - Hashes, arrays, numbers, strings, regular expressions, symbols, nil/true/false,…
  - Use corresponding built-in type for literals (Hash, Array, String, Fixnum, Regexp, etc.) and apply same algorithm as for singleton method completion
  - This time, don’t exclude instance methods

Inherited Method Completion

- Inherited method completion
  - Within a class extending TestCase: as => assert_*
- Completion algorithm
  - Look at the enclosing class to determine supertype
  - Add methods from superclasses and mixins
  - This time, don’t filter out private or protected methods

Variable Completion

- Track variable types in local scope

```
def block_scanf(fstr,&b)
  fs = Scanf::FormatString.new(fstr)
  str = self.dup
  final = []
  begin
    current = str.scanf(fs)
    final.push(yield(current)) unless current.empty?
    str = fs.string_left
  end until current.empty? || str.empty?
  return final
end
```

Tracking Variable Types

- If we see an assignment where we know the expression type, record it
  - x = Foo::Bar.new => x is now of type Foo::Bar
    y = /whatever/ => y is now of type Regexp
    x = foobar() => x is now unknown again
- Completion algorithm
  - Track variables up to the completion point
  - For known types, apply normal instance completion

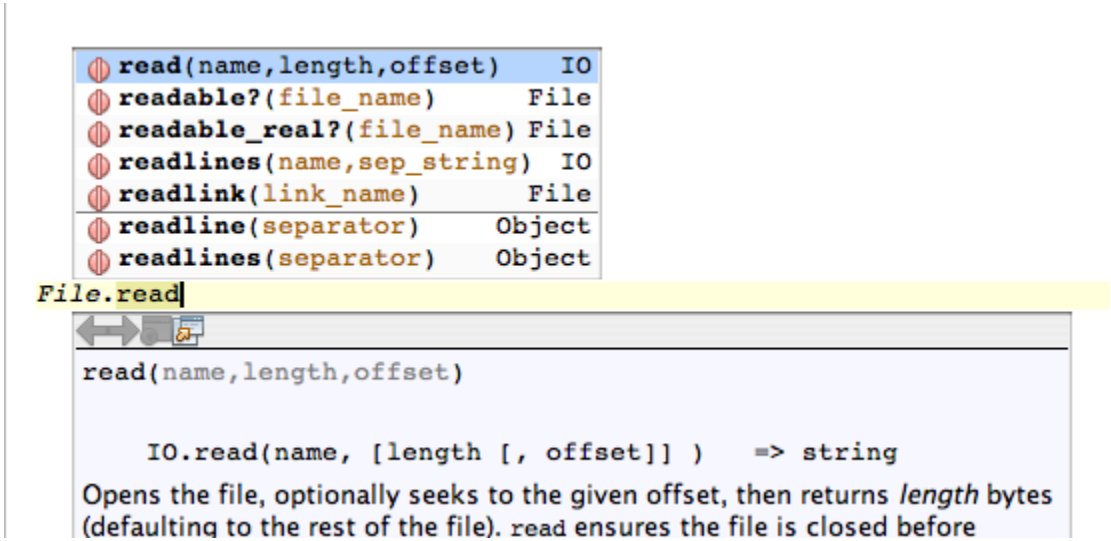
Dealing With Uncertainty

- Works for instance vars, class vars, globals too
  - **Unless intermediate calls have side effects**

```
def block_scanf(fstr,&b)
  @fs = Scanf::FormatString.new(fstr)
  @str = self.dup
  final = []
  begin
    current = str.scanf(@fs)
    final.push(yield(current)) unless current.empty?
    str = @fs.string_left
  end until current.empty? || str.empty?
  return final
end
```

Dealing With Uncertainty

- Let the user decide or interpret the results
  - For example, show best guesses first, then list all matches



Enhanced Variable Tracking

- Compute expression types: x = foo().bar()
- Store return types in the index and use these to compute expression types—when possible
  - Return type is frequently not known
- Suppose String#chomp always returns a String, and String#count always returns a Fixnum:
  x = “foo\n”.chomp().count()
  - We now know that x is of type Fixnum
  - Local variable tracking can proceed

Indexing Return Types

- When indexing code, look for return statements and last expressions
- If the type is known, record it
- This method returns AssertionMessage

```
def build_message(head, template=nil, *arguments)
  template &&= template.chomp
  return AssertionMessage.new(head, template, arguments)
end
```

Multiple Return Types

- A method can have multiple return types
- If the number is small—record all, and let completion include all possibilities
- This method returns {FalseClass,TrueClass}:

```
def has_expires?
  @hash.each do |k, v|
    v.each do |tuple|
      return true if tuple.expires
    end
  end
  false
end
```

Full Class Scope Type Tracking

- Track all assignments to class instance variables and class variables
- If there are no unknown type assignments
  - We can deduce the possible types of the symbol
  - More than one is okay—completion can use a union

```
class Foo
  def whatever(x)
    @state = 0
  end
  def whatever(x)
    @state = "Error:" + x.to_s
  end
  # @state is Fixnum or String
```

Call Site Tracking

- ```
  def make_sound(duck)
        duck.qu^
  end
  ```
- Usually think of "**`duck`**" as unknowable since there is no type information here—any **`duck`** will do
- We are already tracking method references (for refactoring purposes)
- What if we know that **`make_sound`** is called only once—with a known type?
  ```

Call Site Tracking (Cont.)

- More calls yields better accuracy
  - Each additional call type constrains parameter methods
  - Eventually, only “quack” may be present in all call types
  - ```
    def swap(a,b)
      return b,a
    end
    swap(5,10);swap("x","y");swap(/r/,/s/)
    ```
  - The only method available on **a** and **b** above is **to_s** (other than the methods on Object and Class)
  - This approach doesn’t work well when you write methods before calling them (which is common…)
    ```

Parameter Usage Analysis

- We can also look at the actual operations performed on the parameters
  - ```
    def rotate_log(age)
      age.downto(0) do |i|
      ...
    ```
  - Without looking at **any** uses of this method, we can deduce that the type of **`age`** is either **`Integer`** or **`Date`**, the only known classes which provide a **`downto`** method
  - (Unless the program is relying on **`method_missing`**)
    ```

Parameter Usage Analysis (Cont.)

- It's not as simple as it sounds
  - You have to know that the parameter value is still being used at the time of the method reference (back to local variable type tracking)
  - Control flow can introduce complications:

```
def combine(s)
if (foo())
  s.downto(0)
...
```

  - Here we can't conclude that **`s`** is an Integer because the call to foo() could have let other types legally bypass the downto call

Type Hints and Type Assertions

- Before annotations in Java™ code, JavaDoc™ tool markers were used to store metadata for tool use
- We can do the same with Ruby comments
- Record parameter types and return values in the RDoc

FXRuby

```
#
# Draw a circle centered at (_x_, _y_), with radius r.
#
# === Parameters:
#
# +x+:: x-coordinate of the circle's center [Integer]
# +y+:: y-coordinate of the circle's center [Integer]
# +r+:: radius of the circle, in pixels [Integer]
#
# See also #fillCircle.
#
def drawCircle(x, y, r)
  drawArc(x-r, y-r, 2*r, 2*r, 0, 360*64)
end
```

Source: FXRuby-1.6.5/lib/fox16/core.rb from http://www.fxruby.org

“Ruby In Steel” IDE

```
#:return:=>Array
#:arg:names=>Array
#:arg:aName=>String
def addName( names, aName  )
   return names << aName
end
```

Source: http://www.sapphiresteel.com/onlinehelp/Type%20Assertions.html

Type Hints

- We need an agreed upon convention
- Possibly one which allows multiple allowed types, multiple possible return values, or even specific required “quack” methods

Source: http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/c5a19668a0963cb4/085debfba2fe6338

Flame War Alert!

- Type hinting is a highly controversial topic!
- Inserting actual type “constraints” into the source files defeats the purpose of dynamic languages
  - “Sorry, that’s not Ruby!”
- C Ruby relies on comment conventions to document the libraries
  - **/* call-seq:**
    *** Time.at(seconds [, microseconds]) => time**
    ***/ static VALUE time_s_at(argc, argv, klass)**

Source: http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/c5a19668a0963cb4/085debfba2fe6338

Type Hints

- Comments can lie
  - So can type constraints expressed in comments
  - Like comments, type hints express intent, not reality
- IDEs can help detect violations of the type constraints, a la Java code's FindBugs tool
- Parameter hints are not just for type inference
  - Tooltip pop-ups on parameters
  - Improved description of current entry during parameter code completion

Source: http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/c5a19668a0963cb4/085debfba2fe6338

Known Types

- JRuby bridges the Java technology and Ruby worlds
- In the following we know everything about **`frame`** and we can do accurate completion

```
require 'java'

JFrame = javax.swing.JFrame
frame = JFrame.new("Hello Swing")
button = javax.swing.JButton.new("Click Me!")
frame.getContentPane.add button
```

Source: http://groups.google.com/group/comp.lang.ruby/browse_thread/thread/c5a19668a0963cb4/085debfba2fe6338

Recorded Types

- Ruby encourages unit tests with a built-in testing framework (Test::Unit)
- Ruby on Rails takes this even further
- Tools may run unit tests automatically after edits
- With some hooks, unit test execution can record and attribute types for the user program
  - Similar to, and perhaps performed by, code coverage tools
- These are **some** of the parameter's types, not **all**
  - "Sampling"—types may depend on the input data

Agenda

Editors vs. IDEs
Debugging
Code Templates
Code Completion and Type Attribution
Goto Declaration
Refactoring
Demo

Goto Declaration

- Quick navigation (ctrl-click hyperlinks in NetBeans IDE) to declaration point
- Relies on resolving types
- Tricky in Ruby because of “open classes”
  - The “Test::Unit::TestCase” class is defined in many places: test/unit/testcase.rb, active_record/fixtures.rb, action_controller/test_process.rb, action_web_service/test_invoke.rb,...
- “Goto the TestCase Declaration”—which one?

Heuristics

- If the reference includes the module qualifier, we're closer—if we know we're looking for Test::Unit::TestCase we can skip RUNIT::TestCase
- Prefer documented versions of the class over undocumented
  - TestCase in testcase.rb matches; fixtures.rb does not
  - Not enough: File is documented at length in both ftools.rb and the standard Ruby library

Heuristics (Cont.)

- Uses of the class near the reference helps disambiguate—match with methods in each file
  - File.makedirs implies we are looking for File in ftools.rb
- Prefer matches in files that are required
  - Require ‘ftools’
  - May have to look transitively: If resolving “Context”: require ‘irb’ will recursively require ‘irb/context’ which defines IRB::Context
  - May have to look at requires in parent classes as well

Heuristics (Cont.)

- Prefer “built-in” classes over other ones
  - Unless they are defined in the user’s own project files
  - e.g., prefer the built-in “File” over the ftools.rb File
  - Stdlib matching is based on C comment signatures…
- When going to method declarations, additional criteria are available
  - Arity (number of arguments) matching
  - Look in super classes and mixins for matching inherited methods

Arity Matching

- #1: def method(arg1, arg2)
  #2: def method(arg1, arg2, arg3=50)
  #3: def method(arg1, arg2, *arg3)
- **method(1)** matches none of these methods
- **method(x,y)** matches #1, #2 and #3
- **method(x,y,z)** matches #2 and #3
- **method(x,y,z,w)** matches #3
- Used for occurrences highlighting, Goto Decl,…

Other Semantic Editing Features

- Smart selection: select progressively larger surrounding logical code blocks: statement, block, method, class…
- Find usages
  - Including current class—can be split over multiple files
- Semantic highlighting
  - Highlight unused local variables
  - Highlight potentially accidental variable aliasing in blocks
  - Highlight other occurrences of current symbol

Agenda

Editors vs. IDEs
Debugging
Code Templates
Code Completion and Type Attribution
Goto Declaration
Refactoring
Demo

Refactoring

- Rename
  - “Killer app”
  - Complicated by metaprogramming

```
def def_method(mod, methodname, fname='(ERB)')
  mod.module_eval("def #{methodname}\n" + self.src +
    "\nend\n", fname, 0)
end
```

Refactoring

- Local variable/parameter rename
  - Trivial (with a parse tree), and often quite handy
  - Not the same as Search/Replace
    - The third reference to `foo` below is a separate symbol

```
def parse_mode(m)
  result = 0
  (1..9).each do |foo|
    result = 2*result + ((m[foo]==?-) ? 0 : 1)
  end
  foo = count(result)
end
```

Example: Extract Method

- Move a selected chunk of text within a method into its own method
- The IDE looks for local variables used within the block, and converts these into parameters
- The IDE also replaces the selection with a call to the new method
- The IDE may look for other code fragments that can be converted to a call
- New variables created within the block and referenced outside are converted to return values

Ruby Specific Refactorings

- Merge class parts
  - Ruby classes are open and can span many files: this merges them all (or portions of the class) into one
- Extract mixin
  - Move features common to several classes into its own module and add include it as a mixin
- Remove unused scope
  - Remove unnecessary begin/end blocks
- Combine redundant exception handlers
  - Handlers that do the same can be shared

Other Refactorings

- The usual OO-language refactorings also apply
  - Extract Superclass, Convert local variable to a field, Pull Up, Encapsulate Field, Inline Class/Method, move Field/Method/Class, etc.
- Some of these refactorings can take advantage of Ruby techniques
  - Decorator pattern allows us to implement only the unique methods and use a single method_missing to dispatch everything else to the original decorated class
- Ruby-oriented refactoring books in the works

Multi-Language Editing

- Support “languages” like RDoc, Quoted Strings, Regular Expressions, Ruby Within Strings
  - Nested lexing, etc.
- Support RHTML/Eruby: Ruby within HTML markup
  - Full type attribution is necessary such that refactoring, code completion, goto declaration and friends all work
  - More complicated—not well supported anywhere yet*
    - *At the time of this writing, probably obsolete when this is read

DEMO

NetBeans IDE + Ruby Support

Future Directions

- New and better Refactoring operations
- "Smarter" code completion based on improved heuristics
- Static analysis incorporating runtime logs and statistics
  - Perhaps the other way around as well—during a run the debugger can poke the live object hierarchies and perform checks traditionally done by tools like Findbugs
- Incorporate research: "Success Typings" type inference algorithm, etc.

Summary

- Dynamic languages can benefit from features typically available in Java IDEs
- Full-fledged Ruby IDEs in particular are available now
- There is a lot of development in this area

For More Information

- http://wiki.netbeans.org/wiki/view/Ruby
- Ruby related sessions and BOFs:
  - TS-9370: JRuby on Rails: Agility for the Enterprise
  - TS-6503: JRuby; Rails; and Java™ Platform, Enterprise Edition (Java™ EE)
  - TS-9535: Comparing the Developer Experience of Java EE 5.0, Ruby on Rails, and Grails: Lessons Learned
  - TS-9294: Exploiting JRuby: Building Domain-Specific Languages for the Java™ Virtual Machine (JVM™)
  - BOF-9179: Java Platform Web Applications Versus Ruby on Rails: This Time with Tools
  - BOF-2958: Dynamic Scripting Languages BOF
  - BOF-5122: JRubME Is JRuby on Java™ Platform, Micro Edition (Java™ ME)

The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

Q&A

Ruby Tooling: State of the Art

Tor Norbye and Martin Krauskopf

Senior Staff Engineer
Sun Microsystems
http://www.sun.com

Session TS-9972