Ocelot:

A Ruby Compiler

```
int fib(int n){
  if (n<=1) return 1;
  return fib(n-1)+fib(n-2);
}
```

```
def fib(n)
  return 1 if n<=1
  return fib(n-1)+fib(n-2)
end
```

```
def fib(n)
  return 1 if n<=1
  return fib(n-1)+fib(n-2)
end
class FibTest<Test::Unit::TestCase
  def test_fib
    assert_equal 1, fib(0)
    assert_equal 1, fib(1)
    assert_equal 2, fib(2)
    assert_equal 3, fib(3)
    assert_equal 5, fib(4)
    assert_equal 8, fib(5)
  end
end
```

Type Induction

What is a type in ruby?

What is a type in ruby? Wrong answers:

- “Ruby has no types.”
- “Classes are the types.”
- “Singleton classes are the types.”

Type is class+decorators:

```
o=C.new      #o has type C
   ...
o.extend M  #o has type C+M
```

Type is the object's set of name=>method body mappings.

Problems with type inductance:

```
def fib(n)
  return 1 if n<=1
  return fib(n-1)+fib(n-2)
end
class FibTest<Test::Unit::TestCase
  def test_fib
    assert_equal 1, fib(0)
    assert_equal 1, fib(1)
    assert_equal 2, fib(2)
    assert_equal 3, fib(3)
    assert_equal 5, fib(4)
    assert_equal 8, fib(5)
  end
end
```

```ruby
def fib(n)
  return 1 if n<=1
  return fib(n-1)+fib(n-2)
end
class FibTest<Test::Unit::TestCase
  def test_fib
    assert_equal 1, fib(0)
    assert_equal 1, fib(1)
  end
end
```

```
class Cat
  def call
    “meow”
  end
end
```

```
class Dog
  def call
    “bark”
  end
end
```

```
class Bird
  def call
    “tweet”
  end
end
```

```
class Zoo
  def initialize(animals)
    @animals=animals
  end
  def cacophony
    @animals.map{|animal|
      animal.call
    }
  end
end
```

```
class ZooTest<Test::Unit::TestCase
  def test_cacophony
     zoo=Zoo.new([Cat.new, Dog.new])
     zoo.cacophony
  end
end
```

Mocks

~~Mocks~~

Callsite representation:

animal.call

/*animal.call*/

```
(*animal->klass.call)();
```

/*animal.call*/

```
switch(animal->klass){
case Dog:
  Dog_call();
  break;
case Cat:
  Cat_call();
  break;
case Bird:
  Bird_call();
  break;
default:
  warn(“unexpected object type....”);
  rb_funcall(animal,”call”,0);
  break;
}
```

/*animal.call*/

```
switch(animal->klass){
case Dog:
  “bark”;
  break;
case Cat:
  “meow”;
  break;
case Bird:
  “tweet”;
  break;
default:
  warn(“unexpected object type....”);
  rb_funcall(animal,”call”,0);
  break;
}
```

Both compilers and processors can benefit from explicit knowledge of the targets of callsites.

Object Representation:

```
class C
  def initialize(foo,bar)
    @foo,@bar=foo,bar
  end

  def something_else
    @baz=...
  end
end
```

```
class C
  def initialize(foo,bar)
    @foo,@bar=foo,bar
  end

  def something_else
    @baz=...
  end
end
```

```
struct C{
  RObject obj;
}
```

```
struct RObject {
    unsigned long flags;
    VALUE klass;
    struct st_table *iv_tbl;
}
```

```
class C
  def initialize(foo,bar)
    @foo,@bar=foo,bar
  end

  def something_else
    @baz=...
  end
end
```

```
struct C{
  RObject obj;
  VALUE foo;
  VALUE bar;
  VALUE baz;
}
```

```
struct RObject {
    unsigned long flags;
    VALUE klass;
    struct st_table *iv_tbl;
}
```

Binding representation:

```
def m
  a,b,c=1,2,3
end
```

```
def m
  a,b,c=1,2,3
end
```

```
typedef struct m_stackframe{
  struct st_table *locals;
  VALUE a;
  VALUE b;
  VALUE c;
}
```

Hard Stuff

```
def animal.call
  super+”!”
end
```

```
class<<animal
  def call
    super+”?”
  end
end
```

```
module LargeAnimal
  def call
    super.upcase
  end
end
animal.extend LargeAnimal
```

```
/*animal.extend(LargeAnimal)*/
?????;
```

```
/*animal.extend(LargeAnimal)*/
animal->klass=Animal+LargeAnimal;
```

An object's vtable (or klass) field is just a part of its state, and it should be mutable, just like all other state.

```
def method_missing(name,*args)
  ...
end
```

Used in:

- Delegates
- Futures
- RPC proxies

/*animal.call(1)*/

```
switch(animal->klass){
case Dog:
  Dog_call(1);
  break;
case Cat:
  Cat_call(1);
  break;
case Bird:
  Bird_call(1);
  break;
case Delegate:
  Delegate_method_missing(“call”, 1);
  break;
default: ...
}
```

```
eval(some_code)
```

eval(some_code)

(But, almost all evals are static....)

```
#eval(some_code)
if some_code==”foo”
  foo
else
  fail
end
```

```
#eval(some_code)
if some_code==”foo”
  foo
else
  fail    #Could fall back to regular eval here,
          #but refusing to do so is more secure
end
```

Eval

Prescience

a virtuous compiler circle

- Type Induction to nail down the types of receivers.
- Eval Prescience to nail down the arguments to eval.
- These both depend on good test coverage.
- However, poor test coverage can be detected (and logged) at runtime.
- The programmer should use those log statements to discover and plug holes in the tests.
- Which leads to more information for the compiler on the next compile.

Really Hard Stuff

Dynamic eval:

```
while line=gets
  p eval line
end
```

Truly dynamic types:

```
module M1
  def call;1\n”+super end
end
module M2
  def call; “2\n”+super end
end

...
module M20
  def call; “20\n”+super end
end
```

```
Ms=[M1,M2,...M20]
```

```
animal.extend(*Ms.sort_by{rand})
```

The End

- Blog: http://inforadical.net/
- Email: caleb@inforadical.net
- Mailing list: ruby-optimization@googlegroups.com
- ...Questions?
- ...Actively seeking collaborators