TypeCheck

Unknown Author

February 19, 2014

Part I
A Summary of Julia

Julia is a new language designed for technical computing. It is as easy to use and general-purpose as Python, but designed for fast computation, low-level control, and easy to express math. Julia is high-performance, dynamically-typed, and JIT-compiled. It is not focused on new ideas, but on executing existing ideas well, with a focus on being practical and approachable. As a language, some of Julia's distinctive features include multiple dispatch, first-class types, and Lisp-style macros.

1 The Julia Type System

Every value in Julia has a type; variables contain values, but do not themselves have types. Types are arranged into a hierarchy of abstract and concrete types. Abstract types can have subtypes, but cannot be instantiated and do not have properties. Concrete types can be instantiated and have zero or more properties, but cannot have subtypes. Every type has a super type. At top of the hierarchy is the `Any` type; the super type of `Any` is `Any`.

In Julia, types are first-class; types are of type DataType, which is itself of type DataType. Types are used for inference, optimization, dispatch, and documentation, but not for type checking. Because Julia still works (but more slowly) without any type inference, all of the type inference is implemented in the language.

Types can also take parameters; these can be types or `Int`s. For example, `Array{T,N}` is parameterized by the element type and the number of dimensions. Instances of the same type (such as `Array`) with different type parameters (say `Array{Int,6}` and `Array{Number,4}`) are never subtypes of one another.To define a type, you use the `type` keyword:

```
In [1]: type Point{T <: Number}
            x::T
            y::T
        end
```

The Point{T} type will have two properties, `x` and `y`. For any `Point`, the two properties will share a type, and their type will be a subtype of `Number`. From the definition, we also know that when a `Point` is represented in memory, `x` will precede `y`. Julia types are laid out in memory in a way compatible with C structs, which made implementing Julia's C calling functionality easier (and the result more efficient).

2 Introspection in Julia

Julia has admirable introspection and reflection abilities, which are very useful for writing static analysis. For any named function, you can get a type-inferred AST with a simple function call:

In [2]:
```
function foo(x::Int)
    z = x + 5
    return 2 * z
end

code_typed(foo,(Int,))
```

Out [2]:
```
1-element Array{Any,1}:
 :($(Expr(:lambda, {:x}, {{:z},{{:x,Int64,0},{:z,Int64,18}},{}}, quote
# In[2], line 2:
        z = top(box)(Int64,top(add_int)(x::Int64,5))::Int64 # line 3:
        return top(box)(Int64,top(mul_int)(2,z::Int64))::Int64
    end)))
```

The `code_typed` function takes a function and a method signature. Every named function is a generic function: it has one or more methods, each with their own type signature. Julia uses multiple dispatch, which means that it considers the type, number, and order of all the arguments to pick the best match to a call.

`code_typed` returns an untyped `Array` of `Expr`s, the type that represents a node of the Julia AST. For many invocations, this `Array` will only have one element. When the provided signature could match more than one existing method, all possible matches are returned. "Possible" matches occur when you pass an abstract type as part of the signature and some methods of the function accept subtypes of that type. The type of an actual value is always concrete, so the method that would actually get called would vary.

In [3]:
```
e = code_typed(foo,(Int,))[1] #Julia indexes from 1
```

Out [3]:
```
:($(Expr(:lambda, {:x}, {{:z},{{:x,Int64,0},{:z,Int64,18}},{}}, quote
# In[2], line 2:
        z = top(box)(Int64,top(add_int)(x::Int64,5))::Int64 # line 3:
        return top(box)(Int64,top(mul_int)(2,z::Int64))::Int64
    end)))
```

An `Expr` has three fields: `head`, `args`, and `typ`.

In [4]:
```
names(e)
```

Out [4]:
```
3-element Array{Symbol,1}:
 :head
 :args
 :typ
```

- `head` is a symbol indicating the type of expression. For `Expr`s returned by `code_typed`, this will be `:lambda`. (In Julia, `:foo` is a way to write "the symbol `foo`".)
- `typ` is the return type of the method.
- `args` is an `Array` of `Array`s. It contains information about the variables (local, arguments, captured) and body of the function. I'll explain more about the structure of `args` as needed in the rest of this document.

3 Helper Functions

While introspecting on functions is surprisingly easy, there is a lot of ugly code created by the unfortunate structure of the `Expr` type. As a result, I will start by describing a number of helper functions, which will give you a better idea of how it is structured.

3.1 A Function to Retrieve Return Types

`code_typed` returns `Exprs` that have lots of type annotations, including the return type of the function. The outer `Expr` with `head :lambda` will have `typ Any`. The third element of `args` will be another `Expr` with `head :body`. The `typ` property of this Expr will be set to the inferred return type of the function. (There is currently no syntax in Julia to annotate function return types.)

```
In [5]: code_typed(foo,(Int,))[1].args[3].typ
```

```
Out [5]:
        Int64
```

Because this is not especially readable, I wrote a helper function to pull out the return type:

```
In [6]: returntype(e::Expr) =  e.args[3].typ
```

```
Out [6]:
        returntype (generic function with 1 method)
```

Usage examples:

For a call to `code_typed` that we know will have one method returned:

```
In [7]: returntype(code_typed(foo,(Int,))[1])
```

```
Out [7]:
        Int64
```

For calls that might have more than one result:

```
In [8]: Type[returntype(t) for t in code_typed(+,(Number,))]
```

```
Out [8]:
        2-element Array{Type{T<:Top},1}:
         Int64
         Number
```

```
In [9]: map(returntype,code_typed(+,(Any,Any,Any)))
```

```
Out [9]:
        3-element Array{Any,1}:
         BigInt
         BigFloat
         Any
```

3.2 A Function to Retrieve All Expression in the Function Body

The inner `Expr` with head `:body` contains the body of the function: its `args` is an array of `Exprs`. This is another convenience function to make the code more readable.

In [10]:
```
body(e::Expr) = e.args[3].args
```

Out [10]:
```
body (generic function with 1 method)
```

It is used analogously to `returntype` above.

In [11]:
```
body(code_typed(foo,(Int,))[1])
```

Out [11]:
```
4-element Array{Any,1}:
 :( # In[2], line 2:)
 :(z = top(box)(Int64,top(add_int)(x::Int64,5))::Int64)
 :( # line 3:)
 :(return top(box)(Int64,top(mul_int)(2,z::Int64))::Int64)
```

3.3 A Function to Retrieve All Return Statements From a Function

`Exprs` that represent return statements have `head` set to `:return`, so the below function pulls them out of the body.

In [12]:
```
returns(e::Expr) = filter(x-> typeof(x) == Expr && x.head==:return,body(e))
```

Out [12]:
```
returns (generic function with 1 method)
```

In [13]:
```
returns(code_typed(foo,(Int,))[1])
```

Out [13]:
```
1-element Array{Any,1}:
 :(return top(box)(Int64,top(mul_int)(2,z::Int64))::Int64)
```

In [14]:
```
function barr(x::Int)
    x + 2
end

returns(code_typed(barr,(Int,))[1])
```

Out [14]:
```
1-element Array{Any,1}:
 :(return top(box)(Int64,top(add_int)(x::Int64,2))::Int64)
```

Notice that we still get a `:return`, even if we don't use the keyword `return`. The last expression in a function becomes the return value if there is no `return`, and this is expressed in the AST by desugaring to a normal `:return`.

3.4 Other Helper Functions

My project resulted in the `TypeCheck.jl` package for Julia. It is unlikely to be worth the space to explain the implementation of all the other helper functions I wrote here. I'll use functions from `TypeCheck` later, with comments explaining briefly what they do.

Part II

Checking for Stable Return Types

It is good style in Julia for the return type of a method to only depend on the types of the arguments and not on their values. This stability makes behavior more predictable for programmers. It also allows type inference to work better – stable types on called methods allows stable types on the variables you put the return values into.The following method is a simple example of an unstable return type. Sometimes it returns an `Int` and sometimes a `Bool`. The return type of this method would be inferred as `Union(Int64,Bool)`.

In [15]:
```
function unstable(x::Int)
  if x > 5
    return x
  else
    return false
  end
end
```

Out [15]:
```
unstable (generic function with 1 method)
```

In [16]:
```
unstable(5)
```

Out [16]:
```
false
```

In [17]:
```
unstable(1337)
```

Out [17]:
```
1337
```

Until now, there has been no way to automatically check that methods do not behave in this way. Julia's base library is mostly free of this, through the use of code review. While there are instances of instability, they tend to be less obvious – they stem especially from retrieving data from untyped storage, from some interfaces to other environments, or from places where it is necessary (higher-level functions).I have written a static checker to detect that this invariant may be violated. My approach tends more towards false positives than false negatives.

4 Deciding Whether the Return Type is Probably Stable

If the types of an argument to a method are all concrete, then the return type should also be concrete.

4.1 Concrete, Abstract, and Union Types

I've already mentioned concrete and abstract types, which are the leaves and internal nodes of the type hierarchy, respectively. Union types are collection of types. They are similar to abstract types in that they have subtypes, but they do not have names and do not alter the type hierarchy. Union types provide a way to say "Any of these types or their subtypes".

In [18]:
```
[Int, String, Union(Float64,UTF8String)]
```

Out [18]:
```
3-element Array{Type{T<:Top},1}:
 Int64
 String
 Union(UTF8String,Float64)
```

Unlike concrete and abstract types, union types are not represented by `DataType`; they have their own type, `UnionType`.

In [19]:
```
[typeof(x) for x in ans]
```

Out [19]:
```
3-element Array{Type{_},1}:
 DataType
 DataType
 UnionType
```

There is a convenient function for differetiating between concrete types and all other types: `isleaftype`. It returns true for concrete types (the leaves of the type hierarchy) and false for abstract types and union types.

In [20]:
```
isleaftype(Uint128)
```

Out [20]:
```
true
```

In [21]:
```
isleaftype(String)
```

Out [21]:
```
false
```

In [22]:
```
isleaftype(Union(Int,Float32,ASCIIString))
```

Out [22]:
```
false
```

4.2 The Basic Check

Given an `Expr` from `code_typed`, we want to grab the types of the arguments and the return type. If the return type is concrete, then everything is fine: a concrete type can't be unstable. If the return type is not concrete and at least one argument is not concrete, then I don't warn about that method: the cause could be the possible types of the non-concrete argument types.

In [23]:
```
using TypeCheck
function isreturnbasedonvalues(e::Expr)
  rt = returntype(e)
  ts = TypeCheck.argtypes(e) #the type of each argument in e's type signature

  if isleaftype(rt) || rt == None
    return false
  end

  for t in ts
   if !isleaftype(t)
     return false
   end
  end

  return true # return is not concrete type; all args are concrete types
end
```

Out [23]:
```
isreturnbasedonvalues (generic function with 1 method)
```

In [24]:
```
isreturnbasedonvalues(code_typed(unstable,(Int,))[1])
```

Out [24]:
```
true
```

In [25]:
```
isreturnbasedonvalues(code_typed(foo,(Int,))[1])
```

Out [25]:
```
false
```

While this check works on simple examples, it tends to have many false-positives when running on large modules, such as the standard library.

5 Preventing One Unstable Function from Spawning Many More Warnings

With the simple function above, I get a lot of warnings on the base library. When I dug into the causes of some of them, there was a frequent pattern. Many functions would trigger warnings because their return type depended on a call to another function. A small handful of functions would actually need changes to become type-stable, but they are lost in the sea of their users.

These users are sort of "semi-false-positives": their return type is actually unstable, but it's not their fault – and the change probably shouldn't happen there. Most of these functions can be filtered out by looking at the `:returns` in the function body and letting them pass if their return type is determined by `:calls` to other functions (which are unstable).

To be more specific, if `foo` calls `barr`, and `barr` is unstable, then I would like to only warn the user about `barr`, not about `foo`. This simple check will only work if `foo`'s final expression or `return` expression is a call to `barr`. Somewhat surprisingly, this seems to work for basically all of the standard library.

5.1 A Example of Return Type Propogation

Here, `f1` is unstable. We should make a change to `f1` if we want it's return type to be stable. `isreturntypebasedonvalues` would also warn about `f2`. However, `f2` just calls `f1`, so there's not necessarily a change to be made to `f2`.

In [31]:
```
f1(x::Int) = x == 5 ? 42 : pi
returntype(code_typed(f1,(Int,))[1])
```

Out [31]:
```
Union(Int64,MathConst{:})
```

In [32]:
```
f2(y::Int) = f1(y + 2)
returntype(code_typed(f2,(Int,))[1])
```

Out [32]:
```
Union(Int64,MathConst{:})
```

5.2 Preventing Propogation

We can add this to `isreturntypebasedonvalues` by giving failing functions a second chance. This means we can add another check after the loop that looks for non-concrete argument types.

In [41]:
```
function isreturnbasedonvalues(e::Expr)
    rt = returntype(e)
    ts = TypeCheck.argtypes(e)

    if isleaftype(rt) || rt == None
      return false
    end
    for t in ts
     if !isleaftype(t)
       return false
     end
    end

    #a second chance
    #cs is a list of return types for calls in :return exprs in e's body
    cs = [TypeCheck.find_returntype(c,e) for c in TypeCheck.extract_calls_from_returns
    for c in cs
     if rt == c #if e's return type is the same as a call it's making, then it passes
         return false
      end
    end

    return true #e fails the test
  end
```

Out [41]:
```
isreturnbasedonvalues (generic function with 1 method)
```

Above, I use two new helper functions `find_returntype` and `extract_calls_from_returns`. `extract_calls_from_returns` is not expecially interesting; it just examines the insides of `:return` `Exprs` for `:call` `Exprs`. `find_returntype` is more interesting, and I will explain it below.

Results:

```
In [37]: isreturnbasedonvalues(code_typed(foo,(Int,))[1]) #passes
```

```
Out [37]:
false
```

```
In [38]: isreturnbasedonvalues(code_typed(f1,(Int,))[1]) #fails
```

```
Out [38]:
true
```

```
In [40]: isreturnbasedonvalues(code_typed(f2,(Int,))[1]) #passes
```

```
Out [40]:
false
```

6 Determining the Return Type of a `:call`

Here, we start with the output of `extract_calls_from_returns`.

```
In [44]: cs = TypeCheck.extract_calls_from_returns(code_typed(f2,(Int,))[1])
```

```
Out [44]:
1-element Array{Expr,1}:
 :(f1(top(box)(Int64,top(add_int)(y::Int64,2))::Int64)::Union(Int64,Ma
thConst{:}))
```

From this, we need to extract the types of the arguments it is being called with. (We need this to determine which method would be called.)

```
In [46]: cs[1].args[2]
```

```
Out [46]:
:(top(box)(Int64,top(add_int)(y::Int64,2))::Int64)
```

The above gets us the first argument to the `:call`, but it is still a whole `Expr`, not a simple value. Luckily, as you can see at the end, it has been annotated with an inferred type already.

```
In [47]: cs[1].args[2].args[2]
```

```
Out [47]:
:Int64
```

Thus for this example, we know that it is the function named `cs[1].args[1]` being called with `cs[1].args[2].args[2]`.

```
In [48]: println(cs[1].args[1])
         println(cs[1].args[2].args[2])
```

```
f1
Int64
```

However, for this `cs[1]`, the type inference has already provided a return type – `Union(Int64,MathConst{:})`.

In [49]:
```
cs[1].typ
```

Out [49]:
```
Union(Int64,MathConst{:})
```

Since there can be some digging around in the `Expr`s and other `Expr`-like things (`Symbol`s,`Int`s, etc) that occur in `:call` arugments, I wrote `find_returntype` to encapsulate that logic.

In [50]:
```
function find_returntype(e::Expr,context::Expr) #must be :call,:new,:call1
  if Base.is_expr(e,:new); return e.typ; end
  if Base.is_expr(e,:call1) && isa(e.args[1], TopNode); return e.typ; end
  if !Base.is_expr(e,:call); error("Expected :call Expr"); end

  if is_top(e)
    return e.typ
  end

  callee = e.args[1]
  if is_top(callee)
    return find_returntype(callee,context)
  elseif isa(callee,SymbolNode) # only seen (func::F), so non-generic function
    return Any
  elseif is(callee,Symbol)
    if e.typ != Any || any([isa(x,LambdaStaticData) for x in e.args[2:end]])
      return e.typ
    end

    if isdefined(Base,callee)
      f = eval(Base,callee)
      if !isa(f,Function) || !isgeneric(f)
        return e.typ
      end
      fargtypes = tuple([find_argtype(ea,context) for ea in e.args[2:end]])
      return Union([returntype(ef) for ef in code_typed(f,fargtypes)]...)
    else
      return @show e.typ
    end
  end

  return e.typ
end
```

Out [50]:
```
find_returntype (generic function with 1 method)
```

6.1 Deciding If It's Not This Function's Fault

If a function is calling another with concrete types, there will only be one possible method to get called. In that case, it is probably the callee's fault that the return type is bad.

If a function is calling another with looser types, there may be multiple possible methods that could get called. In that case, the way the second function is being used may be causing the problem. If the arguments to the caller are all concrete, where is it getting an abstract type to call this other function?However, if you are just trying to see if one function is type-stable, you might care about `foo`'s unstability, even if it's really `barr`'s fault.

Part III

Stable Types Inside Loops

In Julia, for-loops are generally the fastest way to write code. (Faster than vectorized code; faster than maps or folds.) One way to accidentally decrease their performance is to change the type of a variable in the loop. If all the variables in a loop have stable types, then the code Julia outputs will be the same tight, fast code as a typed, compiled language. If any variable has a type that changes, slower dynamic code will be produced to handle that.It can be easy to write code that has this problem, if you're not aware of it, even in simple programs.

In [26]:
```
x = 5 # x is an Int
for i=1:1000
 x += 100
 x /= 2 # x is a Float64
end
x #x is a Float64
```

Out [26]:
```
100.0
```

In this code example, `x` begins life as an `Int`. In the first iteration of the loop, `x += 100` takes `x` as an `Int` and returns an `Int`; `x /= 2` takes this new `Int` and returns a `Float64`. After this, `x` will be a `Float64` for all the remaining iterations of the loop. This means that the extra dynamic code that is needed to handle `x` being either an `Int` or a `Float64` slows down all the iterations, despite only being needed for the first one. This can be fixed by making `x` a `Float64` from the start: `x = 5.0`.

In [27]:
```
x = 5.0 # x is a Float64
for i=1:1000
 x += 100
 x /= 2 # x is a Float64
end
x #x is a Float64
```

Out [27]:
```
100.0
```

Variables whose types change in loops can be detected in generic functions by looking at the output of `code_typed`. Since loops are lowered to gotos, we need to first find the loops and then check the types of the variables involved. Finding loops can be as simple as looking for gotos that jump backwards in the function: gotos whose labels precede them. Each instruction between the goto and its label is part of the loop body. For each instruction in the loop body, we can look at the inferred type of any variables involved. If the inferred type is a UnionType (or not a leaf type), then the variable's type is unstable.

7 Collecting the Contents of a Loop

Let's begin by defining an example function to work with.

In [38]:
```
function bar(x::Int)
  for i=1:1000
    x += 100
    x /= 2
  end
  x
end
```

Out [38]:
```
bar (generic function with 1 method)
```

The goal is to write a function that takes a method of a generic function and extracts the typed body of any loops. First, we can get the typed body of the function.

In [40]:
```
b = body(bar,(Int,))[1]
```

Out [40]:
```
16-element Array{Any,1}:
 :( # In[38], line 2:)
 :(#s79 = 1)
 :(1: )
 :(unless top(sle_int)(#s79::Int64,1000)::Bool goto 2)
 :(i = #s79::Int64)
 :( # line 3:)
 :(x = +(x::Union(Int64,Float64),100)::Union(Int64,Float64))
 :( # line 4:)
 :(x = /(x::Union(Int64,Float64),2)::Float64)
 :(3: )
 :(#s79 = top(box)(Int64,top(add_int)(1,#s79::Int64))::Int64)
 :(goto 1)
 :(2: )
 :(0: )
 :( # line 6:)
 :(return x::Union(Int64,Float64))
```

The above is an array of `Expr`s and other expression types. We want to find out if, for each `goto` and `unless`, whether their destination label comes before them. Another way to do this is to look at all labels, and check for `goto`s that jump back to each one. While checking to see if we're in a loop, we'll also need to deal with nested loops and knowing when we've finished our loop.

In [44]:
```
inloops = 0
ends = Int[]
loopbody = {}
for i in 1:length(b)
  if typeof(b[i]) == LabelNode
    l = b[i].label
    jumpback = findnext(x -> typeof(x) == GotoNode && x.label == l, b, i)
    if jumpback !=0 #then there's a goto that jumps here
      println("loop from $i to $jumpback")
      push!(ends,jumpback)
      inloops += 1
    end
  end

  if inloops > 0
    println("\t$(b[i])")
    push!(loopbody,b[i])
  end
  if i in ends
    splice!(ends,findfirst(ends,i))
    inloops -= 1
  end

end
```

```
loop from 3 to 12
        :(1: )
        :(unless top(sle_int)(#s79::Int64,1000)::Bool goto 2)
        :(i = #s79::Int64)
        :( # line 3:)
        :(x = +(x::Union(Int64,Float64),100)::Union(Int64,Float64))
        :( # line 4:)
        :(x = /(x::Union(Int64,Float64),2)::Float64)
        :(3: )
        :(#s79 = top(box)(Int64,top(add_int)(1,#s79::Int64))::Int64)
        :(goto 1)
```

We still have a lot of noise here, in the form of line numbers and irrelevant labels.

In [31]:
```
typeof(loopbody[1]) #:(1: )
```

Out [31]:
```
LabelNode
```

In [32]:
```
typeof(loopbody[4]) #:( # line 3:)
```

Out [32]:
```
LineNumberNode
```

In [45]:
```
filter(x -> typeof(x) != LineNumberNode && typeof(x) != LabelNode, loopbody)
```

Out [45]:
```
6-element Array{Any,1}:
 :(unless top(sle_int)(#s79::Int64,1000)::Bool goto 2)
 :(i = #s79::Int64)
 :(x = +(x::Union(Int64,Float64),100)::Union(Int64,Float64))
 :(x = /(x::Union(Int64,Float64),2)::Float64)
 :(#s79 = top(box)(Int64,top(add_int)(1,#s79::Int64))::Int64)
 :(goto 1)
```

This is quite nice. We have the `unless` that contains the exit condition for the for loop, we have the loop variable `i` and `#s79`, and we have the two function calls that modify `x`.

In [34]:
```
  # This is a function for trying to detect loops in a method of a generic function
  # It takes the same arguments as code_typed
  # And returns the lines that are inside one or more loops
  function loopcontents(args...)
    e = code_typed(args...)[1]
    body = e.args[3].args
    loops = Int[]
    nesting = 0
    lines = {}
    for i in 1:length(body)
      if typeof(body[i]) == LabelNode
        l = body[i].label
        jumpback = findnext(x-> typeof(x) == GotoNode && x.label == l, body, i)
        if jumpback != 0
          #println("$i: START LOOP: ends at $jumpback")
          push!(loops,jumpback)
          nesting += 1
        end
      end

      if nesting > 0
        #if typeof(body[i]) == Expr
        #  println("$i: \t", body[i])
        #elseif typeof(body[i]) == LabelNode || typeof(body[i]) == GotoNode
        #  println("$i: ", typeof(body[i]), " ", body[i].label)
        #elseif typeof(body[i]) != LineNumberNode
        #  println("$i: ", typeof(body[i]))
        #end
        push!(lines,(i,body[i]))
      end

      if typeof(body[i]) == GotoNode && in(i,loops)
        splice!(loops,findfirst(loops,i))
        nesting -= 1
        #println("$i: END LOOP: jumps to ",body[i].label)
      end
    end
    lines
  end
```

Out [34]:

```
loopcontents (generic function with 1 method)
```

In [35]:

```
function find_loose_types(arr::Vector)
  lines = ASCIIString[]
  for (i,e) in arr
    if typeof(e) == Expr
      es = copy(e.args)
      while !isempty(es)
        e1 = pop!(es)
        if typeof(e1) == Expr
          append!(es,e1.args)
        elseif typeof(e1) == SymbolNode && !isleaftype(e1.typ) && typeof(e1.typ) ==
          push!(lines,"\t\t$i: $(e1.name): $(e1.typ)")
        end
      end
    end
  end
  isempty(lines) ? lines : unshift!(lines,"")
end
```

Out [35]:

```
find_loose_types (generic function with 1 method)
```

Part IV

Statically Detecting `NoMethodErrors`

The most obviously useful kind of type checking in Julia is to prevent “No Method Error”s. This presents a challenge: methods are often added to generic functions after the fact in Julia, so this checking must be done with as much awareness as possible of the environment in which the call will be made in order to avoid false positives.I have not yet implemented this because I don’t know how to get into the proper context to make the check. I could write first version which ignores this problem.

In [36]:

```
type CallSignature
  name::Symbol
  argtypes::Array{DataType,1}
end
```

In [37]:

```
function find_no_method_errors(args...;mod=None)
  callsigs = find_method_calls(args...)
  output = (Function,CallSignature)[]
  for callsig in callsigs
    f = mod == None ? eval(callsig.name) : eval(mod,callsig.name)
    options = methods(f,tuple(callsig.argtypes...))
    if length(options) == 0
      push!(output,(f,callsig))
    end
  end
  output
end
```

Out [37]:

```
find_no_method_errors (generic function with 1 method)
```

In [38]:

```
function find_method_calls(args...)
  e = code_typed(args...)[1]
  body = e.args[3].args
  lines = CallSignature[]
  for b in body
    if typeof(b) == Expr
      if b.head == :return # want to catch function calls nested in a return
        append!(body,b.args)
      elseif b.head == :call
        if typeof(b.args[1]) == Symbol
          #@show b.args, typeof(b.args[3])
          cs = CallSignature(b.args[1],[typeof(e) == Expr ? e.typ :
            typeof(e) == Symbol ? Any :
            typeof(e) for e in b.args[2:]])
          push!(lines,cs)
        end
      end
    end
  end
  lines
end
```

Out [38]:

```
find_method_calls (generic function with 1 method)
```

In [39]:

```
function aba(x)
  2
end
function aba(x,y::Float64)
  3.14
end
function foo(x)
  aba(x,4)
end
```

Out [39]:

```
foo (generic function with 2 methods)
```

In [42]:

```
c = find_no_method_errors(foo,(Float64,))
```

Out [42]:

```
1-element Array{(Function,CallSignature),1}:
 (aba,CallSignature(:aba,[SymbolNode,Int64]))
```

In [44]:

```
code_typed(foo,(Number,))
```

Out [44]:

```
2-element Array{Any,1}:
 :($(Expr(:lambda, {:x}, {{},{{:x,Int64,0}},{}}, quote  # In[12], line
2:
        unless top(slt_int)(5,x::Int64)::Bool goto 0 # line 3:
        return x::Int64
        goto 1
        0:  # In[12], line 5:
        return false
        1:
    end)))
 :($(Expr(:lambda, {:x}, {{},{{:x,Number,0}},{}}, quote  # In[39],
line 8:
        return aba(x::Number,4)::None
    end)))
```

In []:

```

```