Static Analysis

Leah Hanson

Leah Hanson is a proud alumni of Hacker School and loves helping people learn about Julia. She blogs at http://blog.leahhanson.us/ *and tweets at* @astrieanna.

Introduction

You may be familiar with a fancy IDE that draws red underlines under parts of your code that don't compile. You may have run a linter on your code to check for formatting or style problems. You might run your compiler in super-picky mode with all the warnings turned on. All of these tools are applications of static analysis.

Static analysis is a way to check for problems in your code without running it. "Static" means at compile time rather than at run time, and "analysis" means we're analyzing the code. When you've used the tools I mentioned above, it may have felt like magic. But those tools are just programs—they are made of source code that was written by a person, a programmer like you. In this chapter, we're going to talk about how to implement a couple of static analysis checks. In order to do this, we need to know what we want the check to do and how we want to do it.

We can get more specific about what you need to know by describing the process as three stages:

1. Deciding what you want to check for.

You should be able to explain the general problem you'd like to solve, in terms that a user of the programming language would recognize. Examples include:

- Finding misspelled variable names
- Finding race conditions in parallel code
- Finding calls to unimplemented functions

2. Deciding how exactly to check for it.

While we could ask a friend to do one of the tasks listed above, they aren't specific enough to explain to a computer. To tackle "misspelled variable names", for example, we'd need to decide what misspelled means here. One option would be to claim variable names should be composed of English words from the dictionary; another option is to look for variables that are only used once (the one time you mistyped it).

If we know we're looking for variables that are only used once, we can talk about kinds of variable usages (having their value assigned versus read) and what code would or would not trigger a warning.

3. Implementation details.

This covers the actual act of writing the code, the time spent reading the documentation for libraries you use, and figuring out how to get at the information you need to write the analysis. This could involve reading in a file of code, parsing it to understand the structure, and then making your specific check on that structure.

We're going to work through these steps for each of the individual checks implemented in this chapter. Step 1 requires enough understanding of the language we're analyzing to empathize with the problems its users face. All the code in this chapter is Julia code, written to analyze Julia code.

A Very Brief Introduction to Julia

Julia is a young language aimed at technical computing. It was released at version 0.1 in the spring of 2012; as of the start of 2015, it has reached version 0.3. In general, Julia looks a lot like Python, but with some optional type annotations and without any object-oriented stuff. The feature that most programmers will find novel in Julia is multiple dispatch, which has a pervasive impact on both API design and on other design choices in the language.

Here is a snippet of Julia code:

```
# A comment about incrementfunction increment(x::Int64)
  return x + 1end

increment(5)
```

This code defines a method of the function `increment` that takes one argument, named `x`, of type `Int64`. The method returns the value of `x + 1`. Then, this freshly defined method is called with the value `5`; the function call, as you may have guessed, will evaluate to `6`.

`Int64` is a type whose values are signed integers represented in memory by 64 bits; they are the integers that your hardware understands if your computer has a 64-bit processor. Types in Julia define the representation of data in memory, in addition to influencing method dispatch.

The name `increment` refers to a generic function, which may have many methods. We have just defined one method of it. In many languages, the terms "function" and "method" are used interchangeably; in Julia, they have distinct meanings. This chapter will make more sense if you are careful to understand "function" as a named collection of methods, where a "method" is a specific implementation for a specific type signature.

Let's define another method of the `increment` function:

```
# Increment x by yfunction increment(x::Int64, y::Number)
  return x + yend

increment(5) # => 6
increment(5,4) # => 9
```

Now the function `increment` has two methods. Julia decides which method to run for a given call based on the number and types of the arguments; this is called *dynamic multiple dispatch*:

- **Dynamic** because it's based on the types of the values used at runtime.
- **Multiple** because it looks at the types and order of all the arguments.
- **Dispatch** because this is a way of matching function calls to method definitions.

To put this in the context of languages you may already know, object-oriented languages use single dispatch because they only consider the first argument. (In `x.foo(y)`, the first argument is `x`.)

Both single and multiple dispatch are based on the types of the arguments. The `x::Int64` above is a type annotation purely for dispatch. In Julia's dynamic type system, you could assign a value of any type to `x` during the function without an error.

We haven't really seen the "multiple" part yet, but if you're curious about Julia, you'll have to look that up on your own. We need to move on to our first check.

Checking the Types of Variables in Loops

As in most programming languages, writing very fast code in Julia involves an understanding of how the computer works and how Julia works. An important part of helping the compiler create fast code for you is writing type-stable code; this is important in Julia and JavaScript, and is also helpful in other JIT'd languages. When the compiler can see that a variable in a section of code will always contain the same specific type, the compiler can do more optimizations than if it believes (correctly or not) that there are multiple possible types for that variable. You can read more about why type stability (also called "monomorphism") is important for JavaScript online.

Why This Is Important

Let's write a function that takes an `Int64` and increases it by some amount. If the number is small (less than 10), let's increase it by a big number (50), but if it's big, let's only increase it by 0.5.

```
function increment(x::Int64)
  if x < 10
    x = x + 50
  else
    x = x + 0.5
  end
  return xend
```

This function looks pretty straightforward, but the type of `x` is unstable. I selected two numbers: 50, an `Int64`, and 0.5, a `Float64`. Depending on the value of `x`, it might be added to either one of them. If you add an `Int64` like 22, to a `Float64` like 0.5, you'll get a `Float64` (22.5). Because the type of variable in the function (`x`) could change depending on the value of the arguments to the function (`x`), this method of `increment` and specifically the variable `x` are type-unstable.

`Float64` is a type that represents floating-point values stored in 64 bits; in C, it is called a `double`. This is one of the floating-point types that 64-bit processors understand.

As with most efficiency problems, this issue is more pronounced when it happens during loops. Code inside for loops and while loops is run many, many times, so making it fast is more important than speeding up code that is only run once or twice. Therefore, our first check is to look for variables that have unstable types inside loops.

First, let's look at an example of what we want to catch. We'll be looking at two functions. Each of them sums the numbers 1 to 100, but instead of summing the whole numbers, they divide each one by 2 before summing it. Both functions will get the same answer (2525.0); both will return the same type (`Float64`). However, the first function, `unstable`, suffers from type-instability, while the second one, `stable`, does not.

```
function unstable()
  sum = 0
  for i=1:100
    sum += i/2
  end
  return sumend
```

```
function stable()
  sum = 0.0
  for i=1:100
    sum += i/2
  end
  return sumend
```

The only textual difference between the two functions is in the initialization of `sum`: `sum = 0` versus `sum = 0.0`. In Julia, `0` is an `Int64` literal and `0.0` is a `Float64` literal. How big of a difference could this tiny change make?

Because Julia is Just-In-Time (JIT) compiled, the first run of a function will take longer than subsequent runs. (The first run includes the time it takes to compile the function for these argument types.) When we benchmark functions, we have to be sure to run them once (or precompile them) before timing them.

```
julia> unstable()2525.0

julia> stable()2525.0

julia> @time unstable()
elapsed time: 9.517e-6 seconds (3248 bytes allocated)2525.0

julia> @time stable()
elapsed time: 2.285e-6 seconds (64 bytes allocated)2525.0
```

The `@time` macro prints out how long the function took to run and how many bytes were allocated while it was running. The number of bytes allocated increases every time new memory is needed; it does not decrease when the garbage collector vacuums up memory that's no longer being used. This means that the bytes allocated is related to the amount of time we spend allocating and managing memory, but does not imply that we had all of that memory in use at the same time.

If we wanted to get solid numbers for `stable` versus `unstable` we would need to make the loop much longer or run the functions many times. However, it looks like `unstable` is probably slower. More interestingly, we can see a large gap in the number of bytes allocated; `unstable` has allocated around 3 KB of memory, where `stable` is using 64 bytes.

Since we can see how simple `unstable` is, we might guess that this allocation is happening in the loop. To test this, we can make the loop longer and see if the allocations increase accordingly. Let's make the loop go from 1 to 10000, which is 100 times more iterations; we'll look for the number of bytes allocated to also increase about 100 times, to around 300 KB.

```
function unstable()
  sum = 0
  for i=1:10000
    sum += i/2
  end
  return sumend
```

Since we redefined the function, we'll need to run it so it gets compiled before we measure it. We expect to get a different, larger answer from the new function definition, since it's summing more numbers now.

```
julia> unstable()2.50025e7

julia>@time unstable()
elapsed time: 0.000667613 seconds (320048 bytes allocated)2.50025e7
```

The new `unstable` allocated about 320 KB, which is what we would expect if the allocations are happening in the loop. To explain what's going on here, we're going to look at how Julia works under the hood.

This difference between `unstable` and `stable` occurs because `sum` in `unstable` must be boxed while `sum` in `stable` can be unboxed. Boxed values consist of a type tag and the actual bits that represent the value; unboxed values only have their actual bits. But the type tag is small, so that's not why boxing values allocates a lot more memory.

The difference comes from what optimizations the compiler can make. When a variable has a concrete, immutable type, the compiler can unbox it inside the function. If that's not the case, then the variable must be allocated on the heap, and participate in the garbage collector. Immutable types are a concept specific to Julia. A value of an immutable type can't be changed.

Immutable types are usually types that represent values, rather than collections of values. For example, most numeric types, including `Int64` and `Float64`, are immutable. (Numeric types in Julia are normal types, not special primitive types; you could define a new `MyInt64` that's the same as the provided one.) Because immutable types cannot be modified, you must make a new copy every time you want change one. For example `4 + 6` must make a new `Int64` to hold the result. In contrast, the members of a mutable type can be updated in-place; this means you don't have to make a copy of the whole thing to make a change.

The idea of `x = x + 2` allocating memory probably sounds pretty weird; why would you make such a basic operation slow by making `Int64` values immutable? This is where those compiler optimizations come in: using immutable types doesn't (usually) slow this down. If `x` has a stable, concrete type (such as `Int64`), then the compiler is free to allocate `x` on the stack and mutate `x` in place. The problem is only when `x` has an unstable type (so the compiler doesn't know how big or what type it will be); once `x` is boxed and on the heap, the compiler isn't completely sure that some other piece of code isn't using the value, and thus can't edit it.

Because `sum` in `stable` has a concrete type (`Float64`), the compiler knows that it can store it unboxed locally in the function and mutate its value; `sum` will not be allocated on the heap and new copies don't have to be made every time we add `i/2`.

Because `sum` in `unstable` does not have a concrete type, the compiler allocates it on the heap. Every time we modify sum, we allocated a new value on the heap. All this time spent allocating values on the heap (and retrieving them every time we want to read the value of `sum`) is expensive.

Using `0` versus `0.0` is an easy mistake to make, especially when you're new to Julia. Automatically checking that variables used in loops are type-stable helps programmers get more insight into what the types of their variables are in performance-critical sections of their code.

Implementation Details

We'll need to find out which variables are used inside loops and we'll need to find the types of those variables. We'll then need to decide how to print them in a human-readable format.

- How do we find loops?
- How do we find variables in loops?
- How do we find the types of a variable?
- How do we print the results?
- How do we tell if the type is unstable?

I'm going to tackle the last question first, since this whole endeavour hinges on it. We've looked at an unstable function and seen, as programmers, how to identify an unstable variable, but we need our program to find them. This sounds like it would require simulating the function to look for variables whose values might change—which sounds like it would take some work. Luckily for us, Julia's type inference already traces through the function's execution to determine the types.

The type of `sum` in `unstable` is `Union(Float64,Int64)`. This is a `UnionType`, a special kind of type that indicates that the variable may hold any of a set of types of values. A variable of type `Union(Float64,Int64)` can hold values of type `Int64` or `Float64`; a value can only have one of those types. A `UnionType` joins any number of types (e.g., `UnionType(Float64, Int64, Int32)` joins three types). We're going to look for is `UnionType`d variables inside loops.

Parsing code into a representative structure is a complicated business, and gets more complicated as the language grows. In this chapter, we'll be depending on internal data structures used by the compiler. This means that we don't have to worry about reading files or parsing them, but it does mean we have to work with data structures that are not in our control and that sometimes feel clumsy or ugly.

Besides all the work we'll save by not having to parse the code by ourselves, working with the same data structures that the compiler uses means that our checks will be based on an accurate assessment of the compilers understanding—which means our check will be consistent with how the code actually runs.

This process of examining Julia code from Julia code is called introspection. When you or I introspect, we're thinking about how and why we think and feel. When code introspects, it examines the representation or execution properties of code in the same language (possibly its own code). When code's introspection extends to modifying the examined code, it's called metaprogramming (programs that write or modify programs).

Introspection in Julia

Julia makes it easy to introspect. There are four functions built in to let us see what the compiler is thinking: `code_lowered`, `code_typed`, `code_llvm`, and `code_native`. Those are listed in order of what step in the compilation process their output is from; the first one is closest to the code we'd type in and the last one is the closest to what the CPU runs. For this chapter, we'll focus on `code_typed`, which gives us the optimized, type-inferred abstract syntax tree (AST).

`code_typed` takes two arguments: the function of interest, and a tuple of argument types. For example, if we wanted to see the AST for a function `foo` when called with two `Int64`s, then we would call `code_typed(foo, (Int64,Int64))`.

```
function foo(x,y)
  z = x + y
  return 2 * zend

code_typed(foo,(Int64,Int64))
```

This is the structure that `code_typed` would return:

```
1-element Array{Any,1}:
:($(Expr(:lambda, {:x,:y},
{{:z},{{:x,Int64,0},{:y,Int64,0},{:z,Int64,18}},{}},
 :(begin  # none, line 2:
       z = (top(box))(Int64,(top(add_int))(x::Int64,y::Int64))::Int64 #
line 3:
       return (top(box))(Int64,(top(mul_int))(2,z::Int64))::Int64
   end::Int64))))
```

This is an `Array`; this allows `code_typed` to return multiple matching methods. Some combinations of functions and argument types may not completely determine which method should be called. For example, you could pass in a type like `Any` (instead of `Int64`). `Any` is the type at the top of the type hierarchy; all types are subtypes of `Any` (including `Any`). If we included `Any` in our tuple of argument types, and had multiple matching methods, then the `Array` from `code_typed` would have more than one element in it; it would have one element per matching method.

Let's pull our example `Expr` out to make it easier to talk about.

```
julia> e = code_typed(foo,(Int64,Int64))[1]
:($(Expr(:lambda, {:x,:y},
{{:z},{{:x,Int64,0},{:y,Int64,0},{:z,Int64,18}},{}},
 :(begin  # none, line 2:
       z = (top(box))(Int64,(top(add_int))(x::Int64,y::Int64))::Int64 #
line 3:
       return (top(box))(Int64,(top(mul_int))(2,z::Int64))::Int64
   end::Int64))))
```

The structure we're interested in is inside the `Array`: it is an `Expr`. Julia uses `Expr` (short for expression) to represent its AST. (An abstract syntax tree is how the compiler thinks about the meaning of your code; it's kind of like when you had to diagram sentences in grade school.) The `Expr` we get back represents one method. It has some metadata (about the variables that appear in the method) and the expressions that make up the body of the method.

Now we can ask some questions about `e`.

We can ask what properties an `Expr` has by using the `names` function, which works on any Julia value or type. It returns an `Array` of names defined by that type (or the type of the value).

```
julia> names(e)3-element Array{Symbol,1}:
 :head
 :args
 :typ
```

We just asked `e` what names it has, and now we can ask what value each name corresponds to. An `Expr` has three properties: `head`, `typ` and `args`.

```
julia> e.head
:lambda
```

```
julia> e.typAny

julia> e.args3-element Array{Any,1}:
 {:x,:y}
 {{:z},{{:x,Int64,0},{:y,Int64,0},{:z,Int64,18}},{}}
 :(begin  # none, line 2:
        z = (top(box))(Int64,(top(add_int))(x::Int64,y::Int64))::Int64 #
line 3:
        return (top(box))(Int64,(top(mul_int))(2,z::Int64))::Int64
    end::Int64)
```

We just saw some values printed out, but that doesn't tell us much about what they mean or how they're used.

- `head` tells us what kind of expression this is; normally, you'd use separate types for this in Julia, but `Expr` is a type that models the structure used in the parser. The parser is written in a dialect of Scheme, which structures everything as nested lists. `head` tells us how the rest of the `Expr` is organized and what kind of expression it represents.
- `typ` is the inferred return type of the expression; when you evaluate any expression, it results in some value. `typ` is the type of the value that the expression will evaluate to. For nearly all `Expr`s, this value will be `Any` (which is always correct, since every possible type is a subtype of `Any`). Only the `body` of type-inferred methods and most expressions inside them will have their `typ` set to something more specific. (Because `type` is a keyword, this field can't use that word as its name.)
- `args` is the most complicated part of `Expr`; its structure varies based on the value of `head`. It's always an `Array{Any}` (an untyped array), but beyond that the structure changes.

In an `Expr` representing a method, there will be three elements in `e.args`:

```
julia> e.args[1] # names of arguments as symbols2-element Array{Any,1}:
 :x
 :y
```

Symbols are a special type for representing the names of variables, constants, functions, and modules. They are a different type from strings because they specifically represent the name of a program construct.

```
julia> e.args[2] # three lists of variable metadata3-element
Array{Any,1}:
 {:z}
 {{:x,Int64,0},{:y,Int64,0},{:z,Int64,18}}
 {}
```

The first list above contains the names of all local variables; we only have one (`z`) here. The second list contains a tuple for each variable in and argument to the method; each tuple has the variable name, the variable's inferred type, and a number. The number conveys information about how the variable is used, in a machine- (rather than human-) friendly way. The last list is of captured variable names; it's empty in this example.

```
julia> e.args[3] # the body of the method
:(begin  # none, line 2:
       z = (top(box))(Int64,(top(add_int))(x::Int64,y::Int64))::Int64 #
line 3:
       return (top(box))(Int64,(top(mul_int))(2,z::Int64))::Int64
    end::Int64)
```

The first two `args` elements are metadata about the third. While the metadata is very interesting, it isn't necessary right now. The important part is the body of the method, which is the third element. This is another `Expr`.

```
julia> body = e.args[3]
:(begin  # none, line 2:
       z = (top(box))(Int64,(top(add_int))(x::Int64,y::Int64))::Int64 #
line 3:
       return (top(box))(Int64,(top(mul_int))(2,z::Int64))::Int64
    end::Int64)

julia> body.head
:body
```

This `Expr` has head `:body` because it's the body of the method.

```
julia> body.typInt64
```

The `typ` is the inferred return type of the method.

```
julia> body.args4-element Array{Any,1}:
 :( # none, line 2:)
 :(z = (top(box))(Int64,(top(add_int))(x::Int64,y::Int64))::Int64)
 :( # line 3:)
 :(return (top(box))(Int64,(top(mul_int))(2,z::Int64))::Int64)
```

`args` holds a list of expressions: the list of expressions in the method's body. There are a couple of annotations of line numbers (i.e., `:( # line 3:)`), but most of the body is setting the value of `z` (`z = x + y`) and returning `2 * z`. Notice that these operations have been replaced by `Int64`-specific intrinsic functions. The `top(function-name)` indicates an intrinsic function; something that is implemented in Julia's code generation, rather than in Julia.

We haven't seen what a loop looks like yet, so let's try that.

```
julia> function lloop(x)
         for x = 1:100
           x *= 2
         end
       end
lloop (generic function with 1 method)

julia> code_typed(lloop, (Int,))[1].args[3]
:(begin  # none, line 2:
        #s120 = $(Expr(:new, UnitRange{Int64},
1, :(((top(getfield))(Intrinsics,
         :select_value))((top(sle_int))(1,100)::Bool,100,(top(box))(In
t64,(top(
         sub_int))(1,1))::Int64)::Int64)))::UnitRange{Int64}
        #s119 = (top(getfield))(#s120::UnitRange{Int64},:start)::Int64
unless
         (top(box))(Bool,(top(not_int))(#s119::Int64 ===
(top(box))(Int64,(top(
         add_int))((top(getfield))

(#s120::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool goto 1
        2:
        _var0 = #s119::Int64
        _var1 = (top(box))(Int64,(top(add_int))(#s119::Int64,1))::Int64
        x = _var0::Int64
        #s119 = _var1::Int64 # line 3:
        x = (top(box))(Int64,(top(mul_int))(x::Int64,2))::Int64
        3:
        unless
(top(box))(Bool,(top(not_int))((top(box))(Bool,(top(not_int))
         (#s119::Int64 ===
(top(box))(Int64,(top(add_int))((top(getfield))(

#s120::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool))::Boo
l
         goto 2
        1:        0:
        return
    end::Nothing)
```

You'll notice there's no for or while loop in the body. As the compiler transforms the code from what we wrote to the binary instructions the CPU understands, features that are useful to humans but that are not understood by the CPU (like loops) are removed. The loop has been rewritten as `label` and `goto` expressions. The `goto` has a number in it; each `label` also has a number. The `goto` jumps to the the `label` with the same number.

Detecting and Extracting Loops

We're going to find loops by looking for `goto` expressions that jump backwards.

We'll need to find the labels and gotos, and figure out which ones match. I'm going to give you the full implementation first. After the wall of code, we'll take it apart and examine the pieces.

```
# This is a function for trying to detect loops in the body of a Method#
Returns lines that are inside one or more loopsfunction
loopcontents(e::Expr)
  b = body(e)
  loops = Int[]
  nesting = 0
  lines = {}
  for i in 1:length(b)
    if typeof(b[i]) == LabelNode
      l = b[i].label
      jumpback = findnext(x-> (typeof(x) == GotoNode && x.label == l)
                          || (Base.is_expr(x,:gotoifnot) && x.args[end]
== l),
                        b, i)
      if jumpback != 0
        push!(loops,jumpback)
        nesting += 1
      end
    end
    if nesting > 0
      push!(lines,(i,b[i]))
    end

    if typeof(b[i]) == GotoNode && in(i,loops)
      splice!(loops,findfirst(loops,i))
      nesting -= 1
    end
  end
  linesend
```

And now to explain in pieces:

```
b = body(e)
```

We start by getting all the expressions in the body of method, as an `Array`. `body` is a function that I've already implemented:

```
  # Return the body of a Method.
  # Takes an Expr representing a Method,
  # returns Vector{Expr}.
  function body(e::Expr)
    return e.args[3].args
  end
```

And then:

```
  loops = Int[]
  nesting = 0
  lines = {}
```

`loops` is an `Array` of label line numbers where gotos that are loops occur. `nesting` indicates the number of loops we are currently inside. `lines` is an `Array` of (index, `Expr`) tuples.

```
  for i in 1:length(b)
    if typeof(b[i]) == LabelNode
      l = b[i].label
      jumpback = findnext(
        x-> (typeof(x) == GotoNode && x.label == l)
            || (Base.is_expr(x,:gotoifnot) && x.args[end] == l),
        b, i)
      if jumpback != 0
        push!(loops,jumpback)
        nesting += 1
      end
    end
```

We look at each expression in the body of `e`. If it is a label, we check to see if there is a goto that jumps to this label (and occurs after the current index). If the result of `findnext` is greater than zero, then such a goto node exists, so we'll add that to `loops` (the `Array` of loops we are currently in) and increment our `nesting` level.

```
    if nesting > 0
      push!(lines,(i,b[i]))
    end
```

If we're currently inside a loop, we push the current line to our array of lines to return.

```
    if typeof(b[i]) == GotoNode && in(i,loops)
      splice!(loops,findfirst(loops,i))
      nesting -= 1
    end
  end
  linesend
```

If we're at a `GotoNode`, then we check to see if it's the end of a loop. If so, we remove the entry from `loops` and reduce our nesting level.

The result of this function is the `lines` array, an array of (index, value) tuples. This means that each value in the array has an index into the method-body-`Expr`'s body and the value at that index. Each element of `lines` is an expression that occurred inside a loop.

Finding and Typing Variables

We just finished the function `loopcontents` which returns the `Expr`s that are inside loops. Our next function will be `loosetypes`, which takes a list of `Expr`s and returns a list of variables that are loosely typed. Later, we'll pass the output of `loopcontents` into `loosetypes`.

In each expression that occurred inside a loop, `loosetypes` searches for occurrences of symbols and their associated types. Variable usages show up as `SymbolNode`s in the AST; `SymbolNode`s hold the name and inferred type of the variable.

We can't just check each expression that `loopcontents` collected to see if it's a `SymbolNode`. The problem is that each `Expr` may contain one or more `Expr`; each `Expr` may contain one or more `SymbolNode`s. This means we need to pull out any nested `Expr`s, so that we can look in each of them for `SymbolNode`s.

```
# given `lr`, a Vector of expressions (Expr + literals, etc)# try to find
all occurrences of a variables in `lr`# and determine their typesfunction
loosetypes(lr::Vector)
  symbols = SymbolNode[]
  for (i,e) in lr
    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
          push!(symbols,e1)
        end
      end
    end
  end
  loose_types = SymbolNode[]
  for symnode in symbols
    if !isleaftype(symnode.typ) && typeof(symnode.typ) == UnionType
      push!(loose_types, symnode)
    end
  end
  return loose_typesend
  symbols = SymbolNode[]
  for (i,e) in lr
    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
          push!(symbols,e1)
        end
      end
    end
  end
```

The while loop goes through the guts of all the `Expr`s, recursively. Every time the loop finds a `SymbolNode`, it adds it to the vector `symbols`.

```
loose_types = SymbolNode[]
for symnode in symbols
  if !isleaftype(symnode.typ) && typeof(symnode.typ) == UnionType
    push!(loose_types, symnode)
  end
end
return loose_typesend
```

Now we have a list of variables and their types, so it's easy to check if a type is loose. `loosetypes` does that by looking for a specific kind of non-concrete type, a `UnionType`. We get a lot more "failing" results when we consider all non-concrete types to be "failing". This is because we're evaluating each method with its annotated argument types, which are likely to be abstract.

Making This Usable

Now that we can do the check on an expression, we should make it easier to call on a user's code. We'll create two ways to call `checklooptypes`:

1.

On a whole function; this will check each method of the given function.

2.
3.

On an expression; this will work if the user extracts the results of `code_typed` themselves.

4.

```
## for a given Function, run checklooptypes on each Methodfunction
checklooptypes(f::Callable;kwargs...)
  lrs = LoopResult[]
  for e in code_typed(f)
    lr = checklooptypes(e)
    if length(lr.lines) > 0 push!(lrs,lr) end
  end
  LoopResults(f.env.name,lrs)end
# for an Expr representing a Method,# check that the type of each variable
used in a loop# has a concrete type
checklooptypes(e::Expr;kwargs...) =
 LoopResult(MethodSignature(e),loosetypes(loopcontents(e)))
```

We can see both options work about the same for a function with one method:

```
julia> using TypeCheck


julia> function foo(x::Int)
         s = 0
         for i = 1:x
           s += i/2
         end
         return s
       end
foo (generic function with 1 method)

julia> checklooptypes(foo)
```

```
foo(Int64)::Union(Int64,Float64)
    s::Union(Int64,Float64)
    s::Union(Int64,Float64)
```

```
julia> checklooptypes(code_typed(foo,(Int,))[1])
(Int64)::Union(Int64,Float64)
    s::Union(Int64,Float64)
    s::Union(Int64,Float64)
```

Pretty Printing

I've skipped an implementation detail here: how did we get the results to print out to the REPL?

First, I made some new types. `LoopResults` is the result of checking a whole function; it has the function name and the results for each method. `LoopResult` is the result of checking one method; it has the argument types and the loosely typed variables.

The `checklooptypes` function returns a `LoopResults`. This type has a function called `show` defined for it. The REPL calls `display` on values it wants to display; `display` will then call our `show` implementation.

This code is important for making this static analysis usable, but it is not doing static analysis. You should use the preferred method for pretty-printing types and output in your implementation language; this is just how it's done in Julia.

```
type LoopResult
  msig::MethodSignature
  lines::Vector{SymbolNode}
  LoopResult(ms::MethodSignature,ls::Vector{SymbolNode}) =
new(ms,unique(ls))end
function Base.show(io::IO, x::LoopResult)
  display(x.msig)
  for snode in x.lines
    println(io,"\t",string(snode.name),"::",string(snode.typ))
  endend
type LoopResults
  name::Symbol
  methods::Vector{LoopResult}end
function Base.show(io::IO, x::LoopResults)
  for lr in x.methods
    print(io,string(x.name))
    display(lr)
  endend
```

Looking For Unused Variables

Sometimes, as you're typing in your program, you mistype a variable name. The program can't tell that you meant for this to be the same variable that you spelled correctly before; it sees a variable used only one time, where you might see a variable name misspelled. Languages that require variable declarations naturally catch these misspellings, but many dynamic languages don't require declarations and thus need an extra layer of analysis to catch them.

We can find misspelled variable names (and other unused variables) by looking for variables that are only used once—or only used one way.

Here is an example of a little bit of code with one misspelled name.

```
function foo(variable_name::Int)
  sum = 0
  for i=1:variable_name
    sum += variable_name
  end
  variable_nme = sum
  return variable_nameend
```

This kind of mistake can cause problems in your code that are only discovered when it's run. Let's assume you misspell each variable name only once. We can separate variable usages into writes and reads. If the misspelling is a write (i.e., `worng = 5`), then no error will be thrown; you'll just be silently putting the value in the wrong variable—and it could be frustrating to find the bug. If the misspelling is a read (i.e., `right = worng + 2`), then you'll get a runtime error when the code is run; we'd like to have a static warning for this, so that you can find this error sooner, but you will still have to wait until you run the code to see the problem.

As code becomes longer and more complicated, it becomes harder to spot the mistake—unless you have the help of static analysis.

Left-Hand Side and Right-Hand Side

Another way to talk about "read" and "write" usages is to call them "right-hand side" (RHS) and "left-hand side" (LHS) usages. This refers to where the variable is relative to the `=` sign.

Here are some usages of `x`:

- Left-hand side:

    - `x = 2`
    - `x = y + 22`
    - `x = x + y + 2`
    - `x += 2` (which de-sugars to `x = x + 2`)

- Right-hand side:

    - `y = x + 22`
    - `x = x + y + 2`
    - `x += 2` (which de-sugars to `x = x + 2`)
    - `2 * x`
    - `x`

Notice that expressions like `x = x + y + 2` and `x += 2` appear in both sections, since `x` appears on both sides of the `=` sign.

Looking for Single-Use Variables

There are two cases we need to look for:

1. Variables used once.
2. Variables used only on the LHS or only on the RHS.

We'll look for all variable usages, but we'll look for LHS and RHS usages separately, to cover both cases.

Finding LHS Usages

To be on the LHS, a variable needs to have an `=` sign to be to the left of. This means we can look for `=` signs in the AST, and then look to the left of them to find the relevant variable.

In the AST, an `=` is an `Expr` with the head `:(=)`. (The parentheses are there to make it clear that this is the symbol for `=` and not another operator, `:=`.) The first value in `args` will be the variable name on its LHS. Because we're looking at an AST that the compiler has already cleaned up, there will (nearly) always be just a single symbol to the left of our `=` sign.

Let's see what that means in code:

```
julia> :(x = 5)
:(x = 5)
```

```
julia> :(x = 5).head
:(=)

julia> :(x = 5).args2-element Array{Any,1}:
  :x
 5

julia> :(x = 5).args[1]
:x
```

Below is the full implementation, followed by an explanation.

```
# Return a list of all variables used on the left-hand-side of assignment
(=)## Arguments:#    e: an Expr representing a Method, as from
code_typed## Returns:#    a Set{Symbol}, where each element appears on the
LHS of an assignment in e.#function find_lhs_variables(e::Expr)
  output = Set{Symbol}()
  for ex in body(e)
    if Base.is_expr(ex,:(=))
      push!(output,ex.args[1])
    end
  end
  return outputend
  output = Set{Symbol}()
```

We have a set of Symbols; those are variables names we've found on the LHS.

```
  for ex in body(e)
    if Base.is_expr(ex,:(=))
      push!(output,ex.args[1])
    end
  end
```

We aren't digging deeper into the expressions, because the `code_typed` AST is pretty flat; loops and ifs have been converted to flat statements with gotos for control flow. There won't be any assignments hiding inside function calls' arguments. This code will fail if anything more than a symbol is on the left of the equal sign. This misses two specific edge cases: array accesses (like `a[5]`, which will be represented as a `:ref` expression) and properties (like `a.head`, which will be represented as a `:.` expression). These will still always have the relevant symbol as the first value in their `args`, it might just be buried a bit (as in `a.property.name.head.other_property`). This code doesn't handle those cases, but a couple lines of code inside the `if` statement could fix that.

```
      push!(output,ex.args[1])
```

When we find a LHS variable usage, we `push!` the variable name into the `Set`. The `Set` will make sure that we only have one copy of each name.

Finding RHS usages

To find all the other variable usages, we also need to look at each `Expr`. This is a bit more involved, because we care about basically all the `Expr`s, not just the `:(=)` ones and because we have to dig into nested `Expr`s (to handle nested function calls).

Here is the full implementation, with explanation following.

```
# Given an Expression, finds variables used in it (on right-hand-side)##
Arguments: e: an Expr## Returns: a Set{Symbol}, where each e is used in
a rhs expression in e#function find_rhs_variables(e::Expr)
  output = Set{Symbol}()

  if e.head == :lambda
    for ex in body(e)
      union!(output,find_rhs_variables(ex))
    end
  elseif e.head == :(=)
    for ex in e.args[2:end]  # skip lhs
      union!(output,find_rhs_variables(ex))
    end
  elseif e.head == :return
    output = find_rhs_variables(e.args[1])
  elseif e.head == :call
    start = 2  # skip function name
    e.args[1] == TopNode(:box) && (start = 3)  # skip type name
    for ex in e.args[start:end]
      union!(output,find_rhs_variables(ex))
    end
  elseif e.head == :if
   for ex in e.args # want to check condition, too
     union!(output,find_rhs_variables(ex))
   end
  elseif e.head == :(::)
    output = find_rhs_variables(e.args[1])
  end

  return outputend
```

The main structure of this function is a large if-else statement, where each case handles a different head-symbol.

```
output = Set{Symbol}()
```

`output` is the set of variable names, which we will return at the end of the function. Since we only care about the fact that each of these variables has be read at least once, using a `Set` frees us from worrying about the uniqueness of each name.

```
if e.head == :lambda
  for ex in body(e)
    union!(output,find_rhs_variables(ex))
  end
```

This is the first condition in the if-else statement. A `:lambda` represents the body of a function. We recurse on the body of the definition, which should get all the RHS variable usages in the definition.

```
elseif e.head == :(=)
  for ex in e.args[2:end]  # skip lhs
    union!(output,find_rhs_variables(ex))
  end
```

If the head is `:(=)`, then the expression is an assignment. We skip the first element of `args` because that's the variable being assigned to. For each of the remaining expressions, we recursively find the RHS variables and add them to our set.

```
elseif e.head == :return
  output = find_rhs_variables(e.args[1])
```

If this is a return statement, then the first element of `args` is the expression whose value is returned; we'll add any variables in there into our set.

```
elseif e.head == :call
  # skip function name
  for ex in e.args[2:end]
    union!(output,find_rhs_variables(ex))
  end
```

For function calls, we want to get all variables used in all the arguments to the call. We skip the function name, which is the first element of `args`.

```
elseif e.head == :if
 for ex in e.args # want to check condition, too
   union!(output,find_rhs_variables(ex))
 end
```

An `Expr` representing an if statement has the `head` value `:if`. We want to get variable usages from all the expressions in the body of the if statement, so we recurse on each element of `args`.

```
  elseif e.head == :(::)
    output = find_rhs_variables(e.args[1])
  end
```

The `:(::)` operator is used to add type annotations. The first argument is the expression or variable being annotated; we check for variable usages in the annotated expression.

```
  return output
```

At the end of the function, we return the set of RHS variable usages.

There's a little more code that simplifies the method above. Because the version above only handles `Expr`s, but some of the values that get passed recursively may not be `Expr`s, we need a few more methods to handle the other possible types appropriately.

```
# Recursive Base Cases, to simplify control flow in the Expr version
find_rhs_variables(a) = Set{Symbol}()  # unhandled, should be immediate
val e.g. Int
find_rhs_variables(s::Symbol) = Set{Symbol}([s])
find_rhs_variables(s::SymbolNode) = Set{Symbol}([s.name])
```

Putting It Together

Now that we have the two functions defined above, we can use them together to find variables that are either only read from or only written to. The function that finds them will be called `unused_locals`.

```
function unused_locals(e::Expr)
  lhs = find_lhs_variables(e)
  rhs = find_rhs_variables(e)
  setdiff(lhs,rhs)end
```

`unused_locals` will return a set of variable names. It's easy to write a function that determines whether the output of `unused_locals` counts as a "pass" or not. If the set is empty, the method passes. If all the methods of a function pass, then the function passes. The function `check_locals` below implements this logic.

```
check_locals(f::Callable) = all([check_locals(e) for e in
code_typed(f)])
```

```
check_locals(e::Expr) = isempty(unused_locals(e))
```

Conclusion

We’ve done two static analyses of Julia code—one based on types and one based on variable usages.

Statically-typed languages already do the kind of work our type-based analysis did; additional type-based static analysis is mostly useful in dynamically typed languages. There have been (mostly research) projects to build static type inference systems for languages including Python, Ruby, and Lisp. These systems are usually built around optional type annotations; you can have static types when you want them, and fall back to dynamic typing when you don’t. This is especially helpful for integrating some static typing into existing code bases.

Non-typed-based checks, like our variable-usage one, are applicable to both dynamically and statically typed languages. However, many statically typed languages, like C++ and Java, require you to declare variables, and already give basic warnings like the ones we created. There are still custom checks that can be written; for example, checks that are specific to your project’s style guide or extra safety precautions based on security policies.

While Julia does have great tools for enabling static analysis, it’s not alone. Lisp, of course, is famous for having the code be a data structure of nested lists, so it tends to be easy to get at the AST. Java also exposes its AST, although the AST is much more complicated than Lisp’s. Some languages or language tool-chains are not designed to allow mere users to poke around at internal representations. For open-source tool chains (especially well-commented ones), one option is to add hooks to the enviroment that let you access the AST.

In cases where that won’t work, the final fallback is writing a parser yourself; this is to be avoided when possible. It’s a lot of work to cover the full grammar of most programming languages, and you’ll have to update it yourself as new features are added to the language (rather than getting the updates automatically from upstream). Depending on the checks you want to do, you may be able to get away with parsing only some lines or a subset of language features, which would greatly decrease the cost of writing your own parser.

Hopefully, your new understanding of how static analysis tools are written will help you understand the tools you use on your code, and maybe inspire you to write one of your own.