TypeCheck.jl

Leah Hanson

What is TypeCheck?

- Static analysis to check for problems
- No code changes required to use
- Checks are based on real examples from the mailing list

Implementing Static Analysis

- Decide what to check for
- Decide exactly what to check for
- Implementation details

Implementing Static Analysis

- Decide what to check for
- **Decide exactly what to check for**
- Implementation details

Design Choices

- Only warn when we know something is wrong
- Use Julia’s built-in type inference

Three Checks

- Loop Variable Types
- NoMethodErrors
- Spell Check

Check Loop Variables

- Want variables in loops to have stable types
- Makes a difference in runtime and memory allocation
- Examine the types of variables used in loop to see if they’re stable

NoMethodError

- Runtime error
- Julia's equivalent of a type error

We'd like to predict them statically.

This lets you find problems in little-used code.

Looking for Misspelled Variables

- Sometimes you mistype a variable
- If it's an assignment, no error will be raised
- We can statically find variables:
  - that are only used once
  - that are only assigned
  - that are only read

Check Loop Variables

Check Loop Variables

```
function a()
   sum = 0
   for i=1:100
      sum += i/2
   end
   return sum
end
a()
@time a()
```

```
function b()
   sum = 0.0
   for i=1:100
      sum += i/2
   end
   return sum
end
b()
@time b()
```

Check Loop Variables

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

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

Check Loop Variables

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

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

Check Loop Variables

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

```
julia> a()
2.50025e7

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

julia> code_typed(a,())

1-element Array{Any,1}:

:($(Expr(:lambda, {}, {{:sum,:#s53,:#s52,:#s51,:i,:_var0,:_var1},{{:sum,Any,2},{:#s53,UnitRange{Int64},18},{:#s52,Int64,2},{:#s51,(Int64,Int64),18},{:i,Int64,18},{:_var0,Int64,18},{:_var1,Int64,18}},{}}, :(begin # none, line 2:

sum = 0 # line 3:

#s53 = $(Expr(:new, UnitRange{Int64}, 1, :(top(getfield)(Intrinsics,:select_value)(top(sle_int)(1,10000)::Bool,10000,top(box)(Int64,top(sub_int)(1,1))::Int64)::Int64))::UnitRange{Int64}

#s52 = top(getfield)(#s53::UnitRange{Int64},:start)::Int64

unless top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top(getfield)(#s53::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool goto 1

2:

_var0 = #s52::Int64

_var1 = top(box)(Int64,top(add_int)(#s52::Int64,1))::Int64

i = _var0::Int64

#s52 = _var1::Int64 # line 4:

sum = sum::Union(Int64,Float64) + top(box)(Float64,top(div_float)(top(box)(Float64,top(sitofp)(Float64,i::Int64))::Float64,top(box)(Float64,top(sitofp)(Float64,2))::Float64))::Float64::Float64

3:

unless top(box)(Bool,top(not_int)(top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top(getfield)(#s53::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool))::Bool goto 2

1:

0: # line 6:

return sum::Union(Int64,Float64)

end::Union(Int64,Float64)))))

```
julia> code_typed(a,())[1].args[2][2]
7-element Array{Any,1}:
 {:sum,Any,2}
 {:#s53,UnitRange{Int64},18}
 .....

julia> code_typed(b,())[1].args[2][2]
7-element Array{Any,1}:
 {:sum,Float64,2}
 {:#s53,UnitRange{Int64},18}
 .....
```

julia> code_typed(a,())[1].args[3]

```
:(begin  # none, line 2:
        sum = 0 # line 3:
        #s53 = $(Expr(:new, UnitRange{Int64}, 1, :(top(getfield)(Intrinsics,:select_value)(top(sle_int)(1,10000)::Bool,10000,
top(box)(Int64,top(sub_int)(1,1))::Int64)::Int64)))::UnitRange{Int64}
        #s52 = top(getfield)(#s53::UnitRange{Int64},:start)::Int64
        unless top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top(getfield)(#s53::UnitRange
{Int64},:stop)::Int64,1))::Int64::Bool))::Bool goto 1
        2:
        _var0 = #s52::Int64
        _var1 = top(box)(Int64,top(add_int)(#s52::Int64,1))::Int64
        i = _var0::Int64
        #s52 = _var1::Int64 # line 4:
        sum = sum::Union(Int64,Float64) + top(box)(Float64,top(div_float)(top(box)(Float64,top(sitofp)(Float64,i::Int64))::
Float64,top(box)(Float64,top(sitofp)(Float64,2))::Float64))::Float64::Float64
        3:
        unless top(box)(Bool,top(not_int)(top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top
(getfield)(#s53::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool))::Bool goto 2
        1:
        0:  # line 6:
        return sum::Union(Int64,Float64)
    end::Union(Int64,Float64))
```

julia> code_typed(a,())[1].args[3]

```
:(begin  # none, line 2:
        sum = 0 # line 3:
        #s53 = $(Expr(:new, UnitRange{Int64}, 1, :(top(getfield)(Intrinsics,:select_value)(top(sle_int)(1,10000)::Bool,10000,
top(box)(Int64,top(sub_int)(1,1))::Int64)::Int64)))::UnitRange{Int64}
        #s52 = top(getfield)(#s53::UnitRange{Int64},:start)::Int64
        unless top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top(getfield)(#s53::UnitRange
{Int64},:stop)::Int64,1))::Int64::Bool))::Bool goto 1
        2:
        _var0 = #s52::Int64
        _var1 = top(box)(Int64,top(add_int)(#s52::Int64,1))::Int64
        i = _var0::Int64
        #s52 = _var1::Int64 # line 4:
        sum = sum::Union(Int64,Float64) + top(box)(Float64,top(div_float)(top(box)(Float64,top(sitofp)(Float64,i::Int64))::
Float64,top(box)(Float64,top(sitofp)(Float64,2))::Float64))::Float64::Float64
        3:
        unless top(box)(Bool,top(not_int)(top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top
(getfield)(#s53::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool))::Bool goto 2
        1:
        0:  # line 6:
        return sum::Union(Int64,Float64)
    end::Union(Int64,Float64))
```

julia> code_typed(b,())[1].args[3]

```
:(begin  # none, line 2:
        sum = 0.0 # line 3:
        #s53 = $(Expr(:new, UnitRange{Int64}, 1, :(top(getfield)(Intrinsics,:select_value)(top(sle_int)(1,100)::Bool,100,top
(box)(Int64,top(sub_int)(1,1))::Int64)::Int64)))::UnitRange{Int64}
        #s52 = top(getfield)(#s53::UnitRange{Int64},:start)::Int64
        unless top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top(getfield)(#s53::UnitRange
{Int64},:stop)::Int64,1))::Int64::Bool))::Bool goto 1
        2:
        _var0 = #s52::Int64
        _var1 = top(box)(Int64,top(add_int)(#s52::Int64,1))::Int64
        i = _var0::Int64
        #s52 = _var1::Int64 # line 4:
        sum = top(box)(Float64,top(add_float)(sum::Float64,top(box)(Float64,top(div_float)(top(box)(Float64,top(sitofp)
(Float64,i::Int64))::Float64,top(box)(Float64,top(sitofp)(Float64,2))::Float64))::Float64))::Float64
        3:
        unless top(box)(Bool,top(not_int)(top(box)(Bool,top(not_int)(#s52::Int64 === top(box)(Int64,top(add_int)(top
(getfield)(#s53::UnitRange{Int64},:stop)::Int64,1))::Int64::Bool))::Bool))::Bool goto 2
        1:
        0:  # line 6:
        return sum::Float64
    end::Float64)
```

```
sum = sum::Union(Int64,Float64) + top(box)
(Float64,top(div_float)(top(box)(Float64,top
(sitofp)(Float64,i::Int64))::Float64,top(box)
(Float64,top(sitofp)(Float64,2))::Float64))::
Float64::Float64
```

```
sum = top(box)(Float64,top(add_float)(sum::
Float64,top(box)(Float64,top(div_float)(top(box)
(Float64,top(sitofp)(Float64,i::Int64))::Float64,
top(box)(Float64,top(sitofp)(Float64,2))::
Float64))::Float64))::Float64
```

julia> code_native(a,())

```
        .section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 2
        push    RBP
        mov     RBP, RSP
        push    R15
        push    R14
        push    R13
        push    R12
        push    RBX
        sub     RSP, 56
        mov     QWORD PTR [RBP - 80], 6
Source line: 2
        movabs  RAX, 4308034112
        mov     RCX, QWORD PTR [RAX]
        mov     QWORD PTR [RBP - 72], RCX
        lea     RCX, QWORD PTR [RBP - 80]
        mov     QWORD PTR [RAX], RCX
        mov     QWORD PTR [RBP - 56], 0
        mov     QWORD PTR [RBP - 48], 0
        movabs  RAX, 4328810048
Source line: 2
        mov     QWORD PTR [RBP - 64], RAX
        mov     EBX, 1
        mov     R15D, 10000
Source line: 4
        movabs  R12, 4295395472
        movabs  R13, 4328736592
        movabs  RCX, 4416084224
        movsd   XMM0, QWORD PTR [RCX]
        movsd   QWORD PTR [RBP - 88], XMM0
        movabs  R14, 4295030048
        mov     QWORD PTR [RBP - 56], RAX
        call    R12
        mov     QWORD PTR [RAX], R13
        xorps   XMM0, XMM0
        cvtsi2sd XMM0, RBX
        mulsd   XMM0, QWORD PTR [RBP - 88]
        movsd   QWORD PTR [RAX + 8], XMM0
        mov     QWORD PTR [RBP - 48], RAX
        movabs  RDI, 4362376736
        lea     RSI, QWORD PTR [RBP - 56]
        mov     EDX, 2
        call    R14
Source line: 3
        inc     RBX
Source line: 4
        dec     R15
        mov     QWORD PTR [RBP - 64], RAX
        jne     -70
Source line: 6
        mov     RCX, QWORD PTR [RBP - 72]
        movabs  RDX, 4308034112
        mov     QWORD PTR [RDX], RCX
        add     RSP, 56
        pop     RBX
        pop     R12
        pop     R13
        pop     R14
        pop     R15
        pop     RBP
        ret
```

julia> code_native(b,())

```
        .section        __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 4
        push    RBP
        mov     RBP, RSP
        xorps   XMM0, XMM0
        mov     EAX, 1
        mov     ECX, 100
        movabs  RDX, 4416084592
        movsd   XMM1, QWORD PTR [RDX]
Source line: 4
        xorps   XMM2, XMM2
        cvtsi2sd        XMM2, RAX
        mulsd   XMM2, XMM1
        addsd   XMM0, XMM2
Source line: 3
        inc     RAX
Source line: 4
        dec     RCX
        jne     -28
Source line: 6
        pop     RBP
        ret
```

Running checklooptypes

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

```
julia> checklooptypes(a)
a()::Union(Float64,Int64)
    sum::Union(Float64,Int64)
```

No Method Errors

NoMethodError

- Runtime error
- Julia's equivalent of a type error

We'd like to predict them statically.

This lets you find problems in little-used code.

NoMethodError

```
a(x::Int) = x % 7
b(x::Int) = a(x/2) #NoMethodError
```

This can be found statically.

1. We know /(Int,Int)::Float64
2. We know that a(Float64) is a NoMethodError

Issue in Base (#5923)

```
qrfact(x::Number) =
 QR(fill(one(x), 1, 1), fill(x, 1, 1))
```

Issue in Base (#5923)

```
qrfact(x::Number) =
 QR(fill(one(x), 1, 1), fill(x, 1, 1))


julia> methods(QR)
# 1 method for generic function "QR":
QR{T}(factors::Array{T,2},τ::Array{T,
1})
```

Issue in Base (#5923)

```
qrfact(x::Number) =
 QR(fill(one(x), 1, 1), fill(x, 1, 1))


julia> qrfact(2)
ERROR: no method QR{T}(Array{Int64,2},
Array{Int64,2})
 in qrfact at linalg/factorization.jl:
300
```

Another Issue in Base (#5927)

```
for op in (:+, :-)
    SpecialMatrices = [:Diagonal, :Bidiagonal, :Tridiagonal, :Triangular, :Matrix]
    for (idx, matrixtype1) in enumerate(SpecialMatrices) #matrixtype1 is the sparser matrix type
        for matrixtype2 in SpecialMatrices[idx+1:end] #matrixtype2 is the denser matrix type
            @eval begin #TODO quite a few of these conversions are NOT defined...
                ($op)(A::($matrixtype1), B::($matrixtype2)) = ($op)(convert(($matrixtype2), A), B)
                ($op)(A::($matrixtype2), B::($matrixtype1)) = ($op)(A, convert(($matrixtype2), B))
            end
        end
    end
    for matrixtype1 in (:SymTridiagonal,)                      #matrixtype1 is the sparser matrix type
        for matrixtype2 in (:Tridiagonal, :Triangular, :Matrix) #matrixtype2 is the denser matrix type
            @eval begin
                ($op)(A::($matrixtype1), B::($matrixtype2)) = ($op)(convert(($matrixtype2), A), B)
                ($op)(A::($matrixtype2), B::($matrixtype1)) = ($op)(A, convert(($matrixtype2), B))
            end
        end
    end
    for matrixtype1 in (:Diagonal, :Bidiagonal) #matrixtype1 is the sparser matrix type
        for matrixtype2 in (:SymTridiagonal,)   #matrixtype2 is the denser matrix type
            @eval begin
                ($op)(A::($matrixtype1), B::($matrixtype2)) = ($op)(convert(($matrixtype2), A), B)
                ($op)(A::($matrixtype2), B::($matrixtype1)) = ($op)(A, convert(($matrixtype2), B))
            end
        end
    end
end
```

Another Issue in Base (#5927)

```
for op in (:+, :-)
 for (idx, mt1) in enumerate(SpecialMatrices)
  for mt2 in SpecialMatrices[idx+1:end]
   @eval begin
    ($op)(A::($mt1), B::($mt2)) =
          ($op)(convert(($mt2), A), B)
    ($op)(A::($mt2), B::($mt1)) =
          ($op)(A, convert(($mt2), B))
   end
  end
 end
end
```

Another Issue in Base (#5927)

-(Diagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},Diagonal{T})
-(Bidiagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},Bidiagonal{T})
-(Tridiagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},Tridiagonal{T})
-(SymTridiagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},SymTridiagonal{T})
+(Diagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},Diagonal{T})
+(Bidiagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},Bidiagonal{T})
+(Tridiagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},Tridiagonal{T})
+(SymTridiagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},SymTridiagonal{T})

Another Issue in Base (#5927)

```
-(Diagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},Diagonal{T})
-(Bidiagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},Bidiagonal{T})
-(Tridiagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},Tridiagonal{T})
-(SymTridiagonal{T},Triangular{T<:Number})
-(Triangular{T<:Number},SymTridiagonal{T})
+(Diagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},Diagonal{T})
+(Bidiagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},Bidiagonal{T})
+(Tridiagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},Tridiagonal{T})
+(SymTridiagonal{T},Triangular{T<:Number})
+(Triangular{T<:Number},SymTridiagonal{T})
```

Another Issue in Base (#5927)

```
julia> t = Triangular([1 0 ; 1 0],:U)
2x2 Triangular{Int64}:
 1  0
 0  0

julia> t + t
ERROR: no method +(Triangular{Int64},
Triangular{Int64})
```

Another Issue in Base (#5927)

```
julia> t = Triangular([1 0 ; 1 0],:U)
2x2 Triangular{Int64}:
 1  0
 0  0

julia> d = convert(Diagonal, t)
2x2 Diagonal{Int64}:
 1  0
 0  0
```

Another Issue in Base (#5927)

```
julia> @which d + t
+(A::Diagonal{T},B::Triangular{T<:Number}) at
linalg/special.jl:88

julia> d + t
ERROR: no method +(Triangular{Int64},
Triangular{Int64})
 in + at linalg/special.jl:88
```

Another Issue in Base (#5927)

Not fixed, but a solution would be to define:

+(Triangular, Triangular)
-(Triangular, Triangular)

Another Issue in Base (#5927)

Comment on my Issue:

Thanks for going through all the linear algebra code, by the way. It's no mean feat. -- jiahao

Running checkmethodcalls

```
a(x::Int) = x % 7
b(x::Int) = a(x/2)
```

```
julia>checkmethodcalls(b;mod=Main)
b(Int64)::Any
    a(Float64)
```

It tells you b has a problem, because of the call to a of that signature.

Misspelled Variables

Looking for Misspelled Variables

```
function foo(x)
  answer = 0
  for i in x
    if i % 5 == 0
      anwser += 1
    end
  end
  return answer
end
```

```
function foo(x)
  answer = 0
  for i in x
    if i % 5 == 0
      answer += 1
    end
  end
  return answer
end
```

Looking for Misspelled Variables

```
function foo(x)
  answer = 0
  for i in x
    if i % 5 == 0
      anwser += 1
    end
  end
  return answer
end
```

```
function foo(x)
  answer = 0
  for i in x
    if i % 5 == 0
      answer += 1
    end
  end
  return answer
end
```

Using Variables

- Left Hand Side (LHS)
  - x = 5
- Right Hand Side (RHS)
  - x + 2
  - y = x + 2
- Both
  - x += 2

Checking Usages

- Function that collects a Set of LHS usages
- Function that collects a Set of RHS usages
- Find the difference

Checking Usages

- Function that collects a Set of LHS usages
- Function that collects a Set of RHS usages
- Find the difference

But what about x = x + 2?

Checking Usages

- Function that collects a Set of LHS usages
- Function that collects a Set of RHS usages
- union(lhs, rhs) - intersect(lhs, rhs)

But what about x = x + 2?

Look for variable usages, but only allow x = x + 2 as one usage of x.

Checking Usages

```
function bar(x::Int)
  y = x + 2 + z
  return x
end
```

```
julia> checklocals(bar)
Set{Symbol}({:y,:z})
```

Conclusion

- You can run TypeCheck on your code
- Each function returns a result type, so you can programmatically check that there were no warnings