Detecting Infinite Iteration in Julia Programs Using the `JET.jl` Type Profiling Library

SAMARTH KISHOR, University of Virginia, USA

Julia is a dynamically-typed high-performance programming language that compiles to native code via LLVM [3]. The Julia compiler uses a sophisticated type inference mechanism based on abstract interpretation and data flow analysis to statically generate type information which reduces the amount of inference the JIT compiler has to perform at run-time. `JET.jl` is a static type profiling library which employs Julia's type inference data for bug reports [8]. This report presents an application of `JET.jl` to statically detect the presence of infinite loops in Julia code using type information and inter-procedural analysis of the IR. The inter-procedural control flow analysis on the program IR successfully detected invalid uses of Julia's iteration protocol which would have led to infinite loops in multiple cases.

Additional Key Words and Phrases: static analysis, abstract interpretation, type profiling, infinite iteration

1 INTRODUCTION AND MOTIVATIONS

Julia is a highly dynamic programming language specialized for high-performance computing, but is also widely used as a general-purpose programming language [3]. The Just-In-Time (JIT) compiler requires extensive type information to generate efficient native code via LLVM, and this is best achieved through a combination of ahead-of-time static type inference and run-time dynamic type resolution. Type information is especially important for Julia programs because they often rely heavily on multiple dispatch—a form of polymorphism which is expensive to compute at run-time and relies on specialization to achieve acceptable performance [3]. Since Julia's type system is dynamic, the compiler uses a sophisticated type inference mechanism based on abstract interpretation and data flow analysis to statically generate type information for the expressions in a program. This process can only compute a conservative upper bound on expression types, using the `Any` or `Union{}` type to allow recovery from ill-typed expressions. The static type information generated is typically private to the compiler and has only recently been exposed to library authors to use. This has allowed libraries like `JET.jl` and `Cthulu.jl` to emerge which use this type inference data for bug reports and interactively debugging type inference issues, respectively [5, 6].

2 BACKGROUND AND RELATED WORK

Existing linting tools like `StaticLint.jl` only work with the program's abstract syntax tree (AST) and do not use any type information to advise their analyses [4]. While tools which analyze the AST are useful to the Julia ecosystem, this limits the scope of their analyses to detecting a limited set of errors such as undeclared variables and typos. Julia is dynamically typed but has support for type annotations in code—however, Julia programmers primarily use type annotations in method signatures and avoid them elsewhere which means that the AST does not contain any type information for the vast majority of Julia code. This means that the compiler does not have much data to use while running its type inference system and has to rely on abstract interpretation and data-flow analysis to generate type information.

In contrast to statically-typed languages like OCaml which use unification for type inference, Julia is a dynamically-typed programming language which requires more complex algorithms to generate useful and correct type information with reasonable performance guarantees. Julia's type system is Turing-complete because of the presence of "infinite"

Author's address: Samarth Kishor, sk4gz@virginia.edu, University of Virginia, Charlottesville, Virginia, USA.

```
Base.iterate(nt::NeverTerminate, state = 0) =
    state > nt.val ?
        (state, state + 1) : (state, state + 1)
```

Fig. 1. Incorrect use of the iteration protocol

types such as `Tuple` and `Union` which means that the type convergence algorithm requires heuristics to guarantee termination. Widening guarantees monotonicity and convergence of the type inference routine. An example of widening is transitioning a concrete type to an abstract type more quickly to accelerate convergence. Heuristics can control the length and depth of the widening [8]. The convergence algorithm maintains an acyclic call graph which is good for inlining, resulting in more efficient inter-procedural analyses to generate accurate type information [9].

The abstract interpreter used for type inference operates mostly on the compiler's SSA-form IR which is a flat vector of statements resembling LLVM IR [1]. IR instructions involving expressions are given a type as the abstract interpreter steps through the program and evaluates expressions based on their abstract semantics. The Julia compiler now allows library authors to extend the its built-in methods for their own custom types and workflows. A recent contribution to the compiler "defines a new type, `AbstractInterpreter`, that represents an abstract interpretation pipeline. This `AbstractInterpreter` has a single defined concrete subtype, `NativeInterpreter`, that represents the native Julia compilation pipeline[...] The interpreter object is then threaded throughout most of the type inference pipeline, and allows for straightforward prototyping and replacement of the compiler internals" [10].

`JET.jl` is a library that leverages the abstract interpretation framework exposed by the Julia compiler to perform its own static analyses to statically generate error reports for common bugs such as type errors, incorrect method calls, etc. The library performs the following steps to analyze a Julia program [8]:

- Get the program AST and expand macros
- Execute any "toplevel" actions like type/ generic function/ macro definitions
- Scan each statement of the type lowered form and record the place where an error may occur

`JET.jl` leverages the existing abstract interpretation infrastructure provided by the Julia compiler which greatly simplifies the implementation, removing the need for implementing a separate domain-specific abstract interpreter from scratch. Instead, the library overrides special methods exposed by the Julia compiler for its custom `JETInterpreter` type to perform additional type analysis and generate error reports.

3 CORE CONTRIBUTION

This paper presents an application of the `JET.jl` library to generate error reports for cases of infinite iteration resulting from ill-typed uses of Julia's iteration protocol [2]. Assuming a prior definition of two `structs` named `NeverTerminate` and `CanTerminate`, each containing an `Int` field `val`, Figure 1 illustrates an example of an incorrect use of the iteration protocol which would result in an infinite loop, where the return type of `Base.iterate` would be `Tuple{Int, Int}` instead of the correct `Union{Nothing, Tuple{Int, Int}}`. Figure 2 returns the value `nothing` (similar to `null` or `nil` in other languages) or a tuple of two integers, and thus is an example of a correct use of the iteration protocol.

First, an analysis of the program's typed IR was implemented in order to detect invalid instances of overloaded `Base.iterate` methods, where an invalid method is defined as not having an inferred return type of `Union{Nothing, Tuple{Any, Any}}`.

```
Base.iterate(ct::CanTerminate, state = 0) =
    state > ct.val ?
        nothing : (state, state + 1)
```

Fig. 2. Correct use of the iteration protocol

```
for (var, types) in pairs(iter_types)
    if (!isempty(types) &&
        !any(t ->
             typeintersect(t.typ, Union{Nothing, Tuple{Any, Any}}) !== Bottom,
                          types))
        report!(interp, InfiniteIterationErrorReport(interp, frame))
    end
end
```

Fig. 3. Unsuccessful approach for analyzing the return types of overloaded `Base.iterate` methods

The built-in Julia compiler method `Core.Compiler.typeinf_local` was overloaded to find all the IR statements defining a `Base.iterate` method and determining whether the intersection of the inferred return type and `Union{Nothing, Tuple{Any, Any}}` is equal to `Bottom`, the bottom type in the compiler's abstract interpreter's type lattice. Figure 3 contains part of the code for this approach implemented in the GitHub Pull Request https://github.com/aviatesk/JET.jl/pull/179. This approach was ultimately unsuccessful because it was too permissive and did not report any errors.

The approach suggested and partially implemented by the `JET.jl` library author [7] also involved overloading `Core.Compiler.typeinf_local`, but instead of only analyzing the overloaded `Base.iterate` methods' inferred return types, it performs a control flow analysis which looks for patterns in the lowered IR (see Figure 4), similar to a peephole analysis. The analysis first searches through the basic blocks in the program IR for an `iterate` call followed by a `nothing` check, `Base.not_int` conditional call, and `goto if not` instruction. This series of instructions is specific to an instance of the iteration protocol. Once the pattern is found, the type of the conditional is widened which determines if the value of the `nothing` check is not an `Int`: either true or false. If the condition is not true, then the loop may terminate because it executes the `goto`, otherwise it is possible that it may not terminate.

The issue with this approach is that it was too specific and failed to work on more general examples like the code in Figure 5. I extended this approach to form a more general case that would consider IR patterns without a `Base.not_int` call and also check the inferred types of the `nothing` conditional checks to make sure that the type signature contains a possible `nothing` value in the "true" branch and a tuple in the "else" branch. There were also more specific patterns in the IR which I analyzed as well (see Figures 6 and 7). This combined the first unsuccessful type-based approach with the newer control flow analysis.

4 RESULTS

The combined type- and pattern-based control flow analysis of the program IR was successful in generalizing the infinite iteration error detection approaches to work in all known cases. This analysis also does not break any of the existing unit tests for the library.

```
julia> code_lowered() do
           r = 0
           for i in NeverTerminate(100)
               r += 1
           end
           return r
        end |> first
CodeInfo(
1 □       r = 0
□   %2  = Main.NeverTerminate(100)
□         @_2 = Base.iterate(%2)
□   %4  = @_2 === nothing
□   %5  = Base.not_int(%4)
□□□       goto #4 if not %5
2 □ %7  = @_2
□         i = Core.getfield(%7, 1)
□   %9  = Core.getfield(%7, 2)
□         r = r + 1
□         @_2 = Base.iterate(%2, %9)
□   %12 = @_2 === nothing
□   %13 = Base.not_int(%12)
□□□       goto #4 if not %13
3 □       goto #2
4 □       return r
)
```

Fig. 4. Lowered IR for infinite iteration

REFERENCES

[1] Julia Documentation - Developer Documentation - Julia SSA-form IR. https://docs.julialang.org/en/latest/devdocs/ssair/. (Accessed on 2021-05-03).

[2] Julia Manual - Interfaces - Iteration. https://docs.julialang.org/en/v1/manual/interfaces/#man-interface-iteration. (Accessed on 2021-05-03).

[3] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM Review 59*, 1 (2017), 65–98.

[4] julia vscode. StaticLint.jl - Static Code Analysis for Julia. https://github.com/julia-vscode/StaticLint.jl. (Accessed on 2021-05-05).

[5] JuliaDebug. Cthulu.jl: The slow descent into madness. https://github.com/JuliaDebug/Cthulhu.jl. (Accessed on 2021-05-05).

[6] Kadowaki, S. JET.jl: experimental type checker for Julia. https://github.com/aviatesk/JET.jl. (Accessed on 2021-05-05).

[7] Kadowaki, S. wip: report infinite iteration. https://github.com/aviatesk/JET.jl/pull/186. (Accessed on 2021-05-05).

[8] Kadowaki, S. プログラミング言語 Julia の型推論ルーチンを用いた型プロファイラによる静的なバグ検出について. Bachelor's thesis, Kyoto University, Kyoto, Japan, 2021.

[9] Nash, J. Inference Convergence Algorithm in Julia - Revisited ˉJulia Computing. https://juliacomputing.com/blog/2017/05/15/inference-converage2.html. (Accessed on 2021-05-03).

[10] Saba, E. Add `AbstractInterpreter` to parameterize compilation pipeline. https://github.com/JuliaLang/julia/pull/33955. (Accessed on 2021-05-05).

```
CodeInfo(
1 □      Core.NewvarNode(:(v))
□        y = Base.iterate(itr)
□   %3  = y === Base.nothing
□□□      goto #3 if not %3
2 □      return init
3 □ %6  = Base.getindex(y, 1)
□□□      v = (op)(init, %6)
4 □      goto #8 if not true
5 □ %9  = Base.getindex(y, 2)
□        y = Base.iterate(itr, %9)
□   %11 = y === Base.nothing
□□□      goto #7 if not %11
6 □      goto #8
7 □ %14 = v
□   %15 = Base.getindex(y, 1)
□        v = (op)(%14, %15)
□□□      goto #4
8 □      return v
)
```

Fig. 5. Lowered iteration IR for sum(a for a in NeverTerminate(::Int))

```
# when
#   condt = Core.Compiler.Conditional(:(_), Core.Const(nothing), Tuple{Int64, Int64})
#   t = Core.Const(false)
#   code[cond] = :(_ === Base.nothing)
#   code[cond + 1] = :(goto %_ if not %_)
# check to make sure that the conditional type has
# a Core.Const(nothing) and Tuple{Any, Any}
if t.val === false
    if (condt.vtype === Core.Const(nothing) && isa(condt.elsetype, DataType)
        && condt.elsetype <: Tuple{Any, Any})
        true
    else
        # if the code will return a Const, then it probably will terminate
        isa(ssavaluetypes[dest], Core.Const) && ssavaluetypes[dest] !== NOT_FOUND
    end
```

Fig. 6. Type-based control flow analysis of the generalized iteration IR

```
# when
#   condt = Core.Compiler.Conditional(:(_), Union{}, Core.Const((0, 1)))
#   t = Core.Const(true)
#   code[cond] = :(_ === Base.nothing)
#   code[cond + 1] = :(goto %_ if not %_)
#   code[cond + 2] = :(return _)
# it's okay if the `goto` never occurs because the function
# will return/terminate on the true case
elseif t.val === true && length(code) >= cond + 2 && isa(code[cond + 2], ReturnNode)
    true
```

Fig. 7. Pattern-based control flow analysis of the generalized iteration IR