Summary

"type lattice"
"widening"
" "

1. for dynamic language like Julia,
   type is more for optimizations,

2. Tpye inference is approached as dataflow analysis in Julia.

3. At very high-level. type inference is like running a VM but looking for types instead of values.

4. Guarantee type inference is a challenges →

   ① recursion type inference is a recursive process, but inherently difficult to pause and resume.

   many correctnesse are proposed to solve "cyclic" in the call graph.

   currently TLDR;

   ② static_typeof for comphrehension

Type inference in Julia

https://juliacomputing.com/blog/2016/04/04/inference-convergence.html

Type inference approached as a dataflow problem:

running an intepreter on a program, but only looking at types insteads of values.

↓

can be proven to always converge and terminate.

Basic Algorithms for Type inference.

① statically typed language, flow-insensitive unification-based algorithm → Hindley-Milner

② dynamic typing like Julia → dataflow analysis

↳ variable can have different types at different points in the program

Type inference is used as optimization, rather than to guarantee correctness at compile time.

↳ ① does not need to prove a unique tightest bound on types

② is allowed to widen types heuristically

This is an interesting design :)

```
function sum(list)
    total = 0
    for item in list
            total += item
    end
    return total
end
```

inference execution starts with statements in a function, initialize several states

① program pointer, pc currently active instruction pointers is a list

② types of all variables in the function are set to be Union{ } (aka Bottom), all the type of the corresponding argument.

③ function has an estimated return type of Union{ }

the type inference algorithm then acts like an virtual machine, iterating the following until there are no lines waiting to be examined.

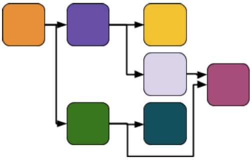
1. take one statement from pc
2. compute side-effects

iteratively apply the current guess to get a guess of the types of of the types of all of the elements of that statements

① assignment → will result in type changing of a variable

② call → looks up or compute the current best guess at the return value of that call, given types of inputs argument

Historical Perspective



(A diagram representing a simple directed call-graph showing all of the function called by `orange`)

① inference is implemented as a ~~recursive descent over the depth-first search.~~

② At each call, inference algorithm pauses inference on current function to recur into callee,

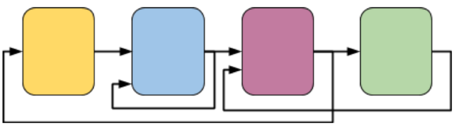
③ Eventually a leaf function is reached

does not make any calls to an uninferred function

④ the whole stack can unravel.

type inference for recursion function →

algorithm needed to take several actions. First, it 1. marked the function as being recursive (`toprec`) and 2. marked every intervening caller as depending on a recursive function (`rec`). Then as the recursion unwound, those functions would record that their return type was an incomplete, under-approximation of the real return type and discard their incomplete inference work (marked `true` in the method's `tfunc` cache). The topmost function in the recursion would then iterate this process until its return type reached a fixed-point.



(A diagram representing a call-graph with several interleaved recursion cycles)

Type inference for Recursion

I mentioned previously that cycles cause issues for the existing type inference algorithm. The root of the problem is that using recursion to solve for the return types makes it inherently difficult to pause and resume type inference to incorporate new information. This impairs computation of the simultaneous convergence of all of the functions involved a cycle and instead led to a solution where only the top-most function in the call-graph is converged at a time. The resulting rework is very time-expensive in cases with heavy recursion, such as running inference on the inference code itself.

Solution is allow pausing and restarting inference on any part of a function at any time.

Type inference for comprehensions

The remaining construct that affects convergence is the `static_typeof` expression. This construct appears in the lowered2 code for a comprehension is an unusual complication in Julia's type inference algorithm. It has required special care to make it look like the result of comprehensions are predictable, even though the current design is not3. That is intended to be fixed soon, so I've relegated an overview of this construct to the footnotes4.

?

Corrected Convergence

The corrected convergence algorithm in PR #15300 works iteratively over all functions in the call graph, building up an implicit graph of the call edges as it comes across them, until all functions have collectively reached a global convergence. When asked

3. next line of the function is added to the list of currently active instruction pointers a list

types of all variables on that next line are merged with the types from the current line, modified by any changes made by the current statement.

This is a complicate process

4. for any if or other control-flow operation (while, goto) the previous operation is performed for all of the possible next statements.

① Given if
② take both if-true and if-false
③ merge results at the end

5. To advoid infinite looping, next instruction is only added if any type of variable is changed

When type inference will be triggered?

How to reach convergence?

① monotonicity → merging two types results in less specific types

② finiteness → limited type number, but Julia does not obey this.

↓

type lattice is forced to have finite height widening

Detecting convergence is handled in several parts.

1. In function
   local convergence at statement level
2. convergence of static_typeof variables
3. global convergence of all functions in the entire call graph