Inference

How inference works

Type inference refers to the process of deducing the types of later values from the types of input values. Julia's approach to inference has been described in blog posts (1, 2).

Debugging compiler.jl

You can start a Julia session, edit `compiler/*.jl` (for example to insert `print` statements), and then replace `Core.Compiler` in your running session by navigating to `base` and executing `include("compiler/compiler.jl")`. This trick typically leads to much faster development than if you rebuild Julia for each change.

Alternatively, you can use the Revise.jl package to track the compiler changes by using the command `Revise.track(Core.Compiler)` at the beginning of your Julia session. As explained in the Revise documentation, the modifications to the compiler will be reflected when the modified files are saved.

A convenient entry point into inference is `typeinf_code`. Here's a demo running inference on `convert(Int, UInt(1))`:

```
# Get the method
atypes = Tuple{Type{Int}, UInt}  # argument types
mths = methods(convert, atypes)  # worth checking that there is only one
m = first(mths)

# Create variables needed to call `typeinf_code`
params = Core.Compiler.Params(typemax(UInt))  # parameter is the world age,
                                              # typemax(UInt) -> most recent
sparams = Core.svec()      # this particular method doesn't have type-parameters
optimize = true            # run all inference optimizations
types = Tuple{typeof(convert), atypes.parameters...} # Tuple{typeof(convert), Type{I
Core.Compiler.typeinf_code(m, types, sparams, optimize, params)
```

If your debugging adventures require a `MethodInstance`, you can look it up by calling `Core.Compiler.specialize_method` using many of the variables above. A `CodeInfo` object may be obtained with

```
# Returns the CodeInfo object for `convert(Int, ::UInt)`:
ci = (@code_typed convert(Int, UInt(1)))[1]
```

The inlining algorithm (inline_worthy)

Much of the hardest work for inlining runs in `inlining_pass`. However, if your question is "why didn't my function inline?" then you will most likely be interested in `isinlineable` and its primary callee, `inline_worthy`. `isinlineable` handles a number of special cases (e.g., critical functions like `next` and `done`, incorporating a bonus for functions that return tuples, etc.). The main decision-making happens in `inline_worthy`, which returns `true` if the function should be inlined.

`inline_worthy` implements a cost-model, where "cheap" functions get inlined; more specifically, we inline functions if their anticipated run-time is not large compared to the time it would take to issue a call to them if they were not inlined. The cost-model is extremely simple and ignores many important details: for example, all `for` loops are analyzed as if they will be executed once, and the cost of an `if...else...end` includes the summed cost of all branches. It's also worth acknowledging that we currently lack a suite of functions suitable for testing how well the cost model predicts the actual run-time cost, although BaseBenchmarks provides a great deal of indirect information about the successes and failures of any modification to the inlining algorithm.

The foundation of the cost-model is a lookup table, implemented in `add_tfunc` and its callers, that assigns an estimated cost (measured in CPU cycles) to each of Julia's intrinsic functions. These costs are based on standard ranges for common architectures (see Agner Fog's analysis for more detail).

We supplement this low-level lookup table with a number of special cases. For example, an `:invoke` expression (a call for which all input and output types were inferred in advance) is assigned a fixed cost (currently 20 cycles). In contrast, a `:call` expression, for functions other than intrinsics/builtins, indicates that the call will require dynamic dispatch, in which case we assign a cost set by `Params.inline_nonleaf_penalty` (currently set at 1000). Note that this is not a "first-principles" estimate of the raw cost of dynamic dispatch, but a mere heuristic indicating that dynamic dispatch is extremely expensive.

Each statement gets analyzed for its total cost in a function called `statement_cost`. You can run this yourself by following the sketch below, where `f` is your function and `tt` is the Tuple-type of the arguments:

```
# A demo on `fill(3.5, (2, 3))`
f = fill
tt = Tuple{Float64, Tuple{Int,Int}}
# Create the objects we need to interact with the compiler
```

```
params = Core.Compiler.Params(typemax(UInt))
mi = Base.method_instances(f, tt)[1]
ci = code_typed(f, tt)[1][1]
opt = Core.Compiler.OptimizationState(mi, params)
# Calculate cost of each statement
cost(stmt::Expr) = Core.Compiler.statement_cost(stmt, -1, ci, opt.sptypes, opt.slott
cost(stmt) = 0
cst = map(cost, ci.code)

# output

31-element Array{Int64,1}:
  0
  0
 20
  4
  1
  1
  1
  0
  0
  0
  ⋮
  0
  0
  0
  0
  0
  0
  0
  0
  0
```

The output is a `Vector{Int}` holding the estimated cost of each statement in `ci.code`. Note that `ci` includes the consequences of inlining callees, and consequently the costs do too.

« Module loading Julia SSA-form IR »

Powered by Documenter.jl and the Julia Programming Language.