Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

Module bindings

The exported names for a `Module` are available using `names(m::Module)`, which will return an array of `Symbol` elements representing the exported bindings. `names(m::Module, all = true)` returns symbols for all bindings in `m`, regardless of export status.

DataType fields

The names of `DataType` fields may be interrogated using `fieldnames`. For example, given the following type, `fieldnames(Point)` returns a tuple of `Symbol`s representing the field names:

```
julia> struct Point
           x::Int
           y
       end

julia> fieldnames(Point)
(:x, :y)
```

The type of each field in a `Point` object is stored in the `types` field of the `Point` variable itself:

```
julia> Point.types
svec(Int64, Any)
```

While `x` is annotated as an `Int`, `y` was unannotated in the type definition, therefore `y` defaults to the `Any` type.

Types are themselves represented as a structure called `DataType`:

```
julia> typeof(Point)
DataType
```

Note that `fieldnames(DataType)` gives the names for each field of `DataType` itself, and one of these fields is the `types` field observed in the example above.

Subtypes

The *direct* subtypes of any `DataType` may be listed using `subtypes`. For example, the abstract `DataType` `AbstractFloat` has four (concrete) subtypes:

```
julia> subtypes(AbstractFloat)
4-element Array{Any,1}:
 BigFloat
 Float16
 Float32
 Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of `subtypes` may be used to inspect the full type tree.

DataType layout

The internal representation of a `DataType` is critically important when interfacing with C code and several functions are available to inspect these details. `isbits(T::DataType)` returns true if `T` is stored with C-compatible alignment. `fieldoffset(T::DataType, i::Integer)` returns the (byte) offset for field *i* relative to the start of the type.

Function methods

The methods of any generic function may be listed using `methods`. The method dispatch table may be searched for methods accepting a given type using `methodswith`.

Expansion and lowering

As discussed in the Metaprogramming section, the `macroexpand` function gives the unquoted and interpolated expression (`Expr`) form for a given macro. To use `macroexpand`, `quote` the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

```
julia> macroexpand(@__MODULE__, :(@edit println("")) )
:(InteractiveUtils.edit(println, (Base.typesof)("")))
```

The functions `Base.Meta.show_sexpr` and `dump` are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the `Meta.lower` function gives the `lowered` form of any expression and is of particular interest for understanding how language constructs map to primitive operations such as assignments, branches, and calls:

```
julia> Meta.lower(@__MODULE__, :( [1+2, sin(0.5)] ))
:($(Expr(:thunk, CodeInfo(
    @ none within `top-level scope'
1 — %1 = 1 + 2
│   %2 = sin(0.5)
│   %3 = Base.vect(%1, %2)
└──      return %3
))))
```

Intermediate and compiled representations

Inspecting the lowered form for functions requires selection of the specific method to display, because generic functions may have many methods with different type signatures. For this purpose, method-specific code-lowering is available using `code_lowered`, and the type-inferred form is available using `code_typed`. `code_warntype` adds highlighting to the output of `code_typed`.

Closer to the machine, the LLVM intermediate representation of a function may be printed using by `code_llvm`, and finally the compiled machine code is available using `code_native` (this will trigger JIT compilation/code generation for any function which has not previously been called).

For convenience, there are macro versions of the above functions which take standard function calls and expand argument types automatically:

```
julia> @code_llvm +(1,1)

define i64 @"julia_+_130862"(i64, i64) {
top:
    %2 = add i64 %1, %0
    ret i64 %2
}
```

For more informations see `@code_lowered`, `@code_typed`, `@code_warntype`, `@code_llvm`, and `@code_native`.

Printing of debug information

The aforementioned functions and macros take the keyword argument `debuginfo` that controls the level debug information printed.

```
julia> @code_typed debuginfo=:source +(1,1)
CodeInfo(
    @ int.jl:53 within `+'
1 — %1 = Base.add_int(x, y)::Int64
└──      return %1
) => Int64
```

Possible values for `debuginfo` are: `:none`, `:source`, and `:default`. Per default debug information is not printed, but that can be changed by setting `Base.IRShow.default_debuginfo[] = :source`.

« Unicode Initialization of the Julia runtime »