Computational topology: Lecture 13

Alberto Paoluzzi

May 3, 2019

1. Symbolic debugger
2. Symbolic debugging in Julia
3. Rebugger tutorial
4. Restart our debugging session

Symbolic debugger

Introduction

I can't recommend too strongly that you learn how to use a debugger

- If your programs assemble and link properly but do not work, you will remain in the dark about the reason for this, until you watch them working under debugger control
- Often, one debugging session will show up an error instantly and can save hours of time trying out various different things

What is a debugger?

A debugger is able to run another program (the "debuggee") in closely controlled conditions

- This enables you to single step through the debuggee's code: the processor will execute a single instruction at a time, and you can watch the effect of this on the debuggee's registers and flags, stack and memory areas
- Usually a debugger allows you to choose whether to trace into CALLS or execute but jump over them
- A debugger will also allow you to set breakpoints where you can stop execution and proceed from there, or to run the debuggee until something goes wrong.

What is a symbolic debugger?

A symbolic debugger knows the addresses of the symbols and is able to display them in the disassembly.

- Data references in the disassembly are shown by data label
- A symbolic debugger may also use the code labels to allow the user to establish breakpoints, and may display the contents of memory by reference to data labels
- The symbols are known to the debugger either because the symbol information is embedded in the executable, or kept in a separate file
- This is done at link-time and is achieved by the linker, which if asked, will sort the labels in the object files and put them in the executable file (or in a separate file) to be read by the debugger as symbols.

Retrace the steps prior to the bug

Try to note the sequence of events leading up to the bug each time the fault occurs

- Then carry out that sequence again to check that the fault then occurs
- Try to isolate the fault by removing some of the steps or by taking other steps
- Get the sequence as short as you can
- This process will help to find the most likely culprit for the fault, and reduce the procedures that you need to check

Symbolic debugging in Julia

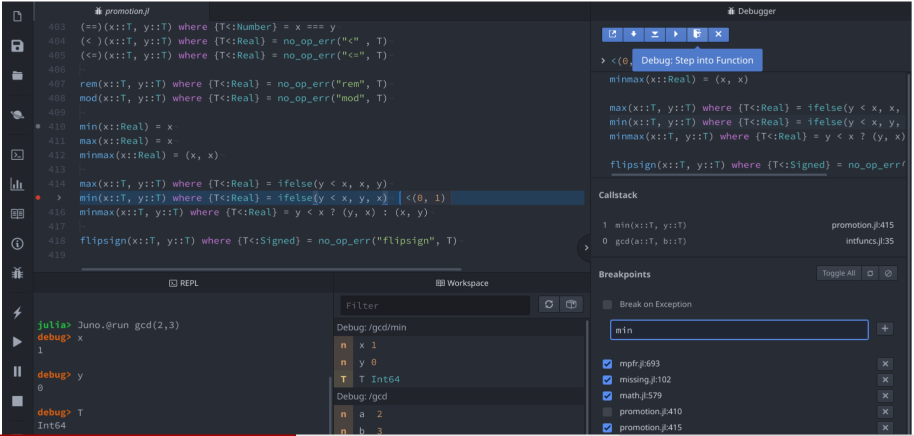
A Julia interpreter and debugger

A Julia interpreter and debugger

- Step into functions and manually walk through your code while inspecting its state
- Set breakpoints and trap errors, allowing you to discover what went wrong at the point of trouble
- Interactively update and replace existing code to rapidly fix bugs in place without restarting
- Use the full-featured IDE in Juno to bundle all these features together in an easy to use graphical interface

Juno

- The `Juno.@run` macro interprets your code and drops you in a debugging session if it hits a breakpoint
- `Juno.@enter` allows you to step through starting from the first line.



Debugger and Rebugger

If you have a different favorite editor than Atom—or via remote sessions through a console—you can alternatively perform debugging via the REPL

- There are two REPL interfaces:
  - Debugger offers a "step, next, continue" interface similar to debuggers like gdb,
  - Rebugger aims to provide a console interface that is reminiscent of an IDE

Debugger and Rebugger

If you have a different favorite editor than Atom—or via remote sessions through a console—you can alternatively perform debugging via the REPL

- There are two REPL interfaces:
  - Debugger offers a "step, next, continue" interface similar to debuggers like gdb,
  - Rebugger aims to provide a console interface that is reminiscent of an IDE

- Debugger has some capabilities that none of the other interfaces offer, so it should be your choice for particularly difficult cases

Debugger session example

```
julia> @enter closestpair([[0, -0.3], [1., 1.], [1.5, 2], [2, 2], [3, 3]])
In closestpair(P) at /mnt/c/Users/Kristoffer/Debugging/closest_pair.jl
>4  N = length(P)
 5  if N < 2 return (Inf, ()) end
 6  mindst = norm(P[1] - P[2])
 7  minpts = (P[1], P[2])
●8  for i in 1:N-1, j in i+1:N

About to run: (length)(Array{Float64,1}[[0.0, -0.3], [1.0, 1.0], [1.5, 2.0], [2.0, 2.0], [3.0, 3.0]])
1|debug> c
Hit breakpoint:
In closestpair(P) at /mnt/c/Users/Kristoffer/Debugging/closest_pair.jl
  4  N = length(P)
  5  if N < 2 return (Inf, ()) end
  6  mindst = norm(P[1] - P[2])
  7  minpts = (P[1], P[2])
> 8  for i in 1:N-1, j in i+1:N
  9      tmpdst = norm(P[i] - P[j])
 10      if tmpdst < mindst
 11          mindst = tmpdst
 12          minpts = (P[i], P[j])

About to run: (-)(5, 1)
1|debug> fr
[1] closestpair(P) at /mnt/c/Users/Kristoffer/Debugging/closest_pair.jl
  | P::Array{Array{Float64,1},1} = Array{Float64,1}[[0.0, -0.3], [1.0, 1.0], [1.5, 2.0], [2.0, 2.0], [3.0, 3.0]]
  | N::Int64 = 5
  | mindst::Float64 = 1.6401219466856727
  | minpts::Tuple{Array{Float64,1},Array{Float64,1}} = ([0.0, -0.3], [1.0, 1.0])
  | T::DataType = Float64
1|julia> norm(P[2] - P[1])
1.6401219466856727
```

Figure 2: Some examples

Rebugger

Rebugger enters calls via a key binding

- To try it, type `gcd(10, 20)` and without hitting enter type `Meta-i` (`Esc-i`, `Alt-i`, or `option-i`)
- After a short pause the display should update; type ? to see the possible actions:

JuliaInterpreter

Contains the logic needed to evaluate and inspect running Julia code

- An interpreter lends itself naturally to step-wise code evaluation and the implementation of breakpoints

```
using JuliaInterpreter
A = rand(1:10, 5)
@interpret sum(A)
```

JuliaInterpreter gained the ability to interpret "top-level code", for example the code used to define packages and create test suites

JuliaInterpreter

JuliaInterpreter gained support for breakpoints

- While not strictly a feature of interpreters, they are necessary to build a capable debugger and can be viewed as an additional form of control-flow within the interpreter itself
- These breakpoints can be set manually with functions `breakpoint` and a macro `@breakpoint`, manipulated in Juno, Rebugger, or Debugger, or added directly to code with the `@bp` macro
- Existing breakpoints can be disabled, enabled, or removed
- We support setting of breakpoints at specific source lines or on entry to a specific method, conditional and unconditional breakpoints, and can automatically trap errors as if they were manually-set breakpoints

CodeTracking

`CodeTracking` won't do anything useful unless the user is also running `Revise`, because Revise will be responsible for updating CodeTracking's internal variables

CodeTracking

`CodeTracking` won't do anything useful unless the user is also running `Revise`, because Revise will be responsible for updating CodeTracking's internal variables

However, Revise is a fairly large (and fairly complex) package, and currently it's not easy to discover how to extract particular kinds of information from its internal storage

- CodeTracking is designed to be the new "query" part of Revise.jl
- The aim is to have a very simple API that developers can learn in a few minutes and then incorporate into their own packages; its lightweight nature means that they potentially gain a lot of functionality without being forced to take a big hit in startup time

Rebugger tutorial

Rebugger tutorial

Rebugger is an expression-level debugger for Julia

- It has no ability to interact with or manipulate call stacks (see Gallium), but it can trace execution via the manipulation of Julia expressions

The name "Rebugger" has 3 meanings:

1. it is a REPL-based debugger (more on that in the documentation)
2. it is the Revise-based debugger
3. it supports repeated-execution debugging

Rebugger tutorial

clone locally the package https://github.com/timholy/Rebugger.jl
compile the `docs`
start from `build/index.html`

Restart our debugging session

Our 3D test example: here is OK

```
using LinearAlgebraicRepresentation, Plasm
Lar = LinearAlgebraicRepresentation

function twocubes()
    #V,(VV,EV,FV,CV) = Lar.cuboid([0.5,0.5,0.5],true,[-0.5,-0.5,-0.5])
    V,(VV,EV,FV,CV) = Lar.cuboidGrid([1,1,1],true)
    mybox = (V,CV,FV,EV)

    #twocubes = Lar.Struct([ mybox , Lar.t(0.3,0.4,0.5), Lar.r(pi/5,0,0), Lar.r(0,0,pi/12), mybox ])
    twocubes = Lar.Struct([ mybox , Lar.t(0.3,0.4,0.5), Lar.r(pi/3,0,0), Lar.r(0,0,pi/6), mybox ])
    V,CV,FV,EV = Lar.struct2lar(twocubes)
    Plasm.view(V,CV)

    cop_EV = Lar.coboundary_0(EV::Lar.Cells);
    cop_EW = convert(Lar.ChainOp, cop_EV);
    cop_FE = Lar.coboundary_1(V, FV::Lar.Cells, EV::Lar.Cells);
    W = convert(Lar.Points, V');

    V, copEV, copFE, copCF = Lar.Arrangement.spatial_arrangement( W::Lar.Points, cop_EW::Lar.ChainOp, cop_FE::L
    EV = Lar.cop2lar(copEV)
    FE = [findnz(copFE[k,:])[1] for k=1:size(copFE,1)]
    FV = [collect(Set(cat(EV[e] for e in FE[f]))) for f=1:length(FE)]
    FV = convert(Lar.Cells, FV)
    W = convert(Lar.Points, V')

    Plasm.view(Plasm.numbering(0.25)((W,[[[k] for k=1:size(W,2)],EV,FV])))

    triangulated_faces = Lar.triangulate(V, [copEV, copFE])
    FVs = convert(Array{Lar.Cells}, triangulated_faces)
    V = convert(Lar.Points, V')
    Plasm.viewcolor(V::Lar.Points, FVs::Array{Lar.Cells})
```

Our 3D test example: here is KO . . .

```
using LinearAlgebraicRepresentation, Plasm
Lar = LinearAlgebraicRepresentation

function twocubes()
    V,(VV,EV,FV,CV) = Lar.cuboid([0.5,0.5,0.5],true,[-0.5,-0.5,-0.5])
    #V,(VV,EV,FV,CV) = Lar.cuboidGrid([1,1,1],true)
    mybox = (V,CV,FV,EV)

    #twocubes = Lar.Struct([ mybox , Lar.t(0.3,0.4,0.5), Lar.r(pi/5,0,0), Lar.r(0,0,pi/12), mybox ])
    twocubes = Lar.Struct([ mybox , Lar.t(0.3,0.4,0.5), Lar.r(pi/3,0,0), Lar.r(0,0,pi/6), mybox ])
    V,CV,FV,EV = Lar.struct2lar(twocubes)
    Plasm.view(V,CV)
```

Loops. . .

```
sigma = 67
sigma = 68
sigma = 69
sigma = 70
sigma = 71
sigma = 72
0%
```

aaaaaa

aaaaaa