Intros + Lambda Calculus

CS 130 sp 20

4/3/20

Agenda

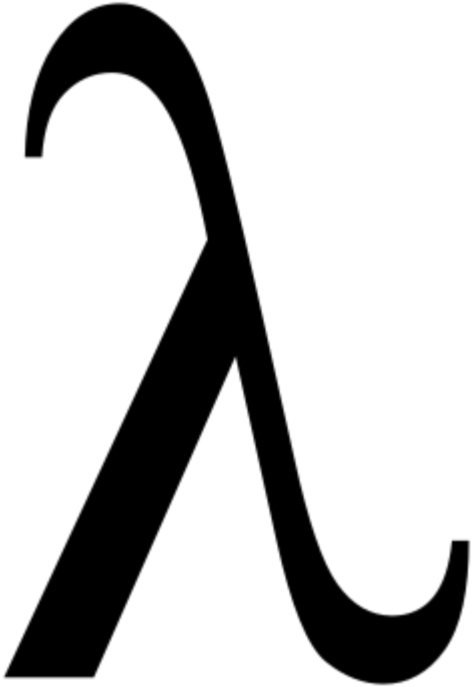
Setup

What is the lambda calculus

Syntax

Alpha/Beta reductions

PA0 tips



Agenda

Setup

What is the lambda calculus

Syntax

Alpha/Beta reductions

PA0 tips

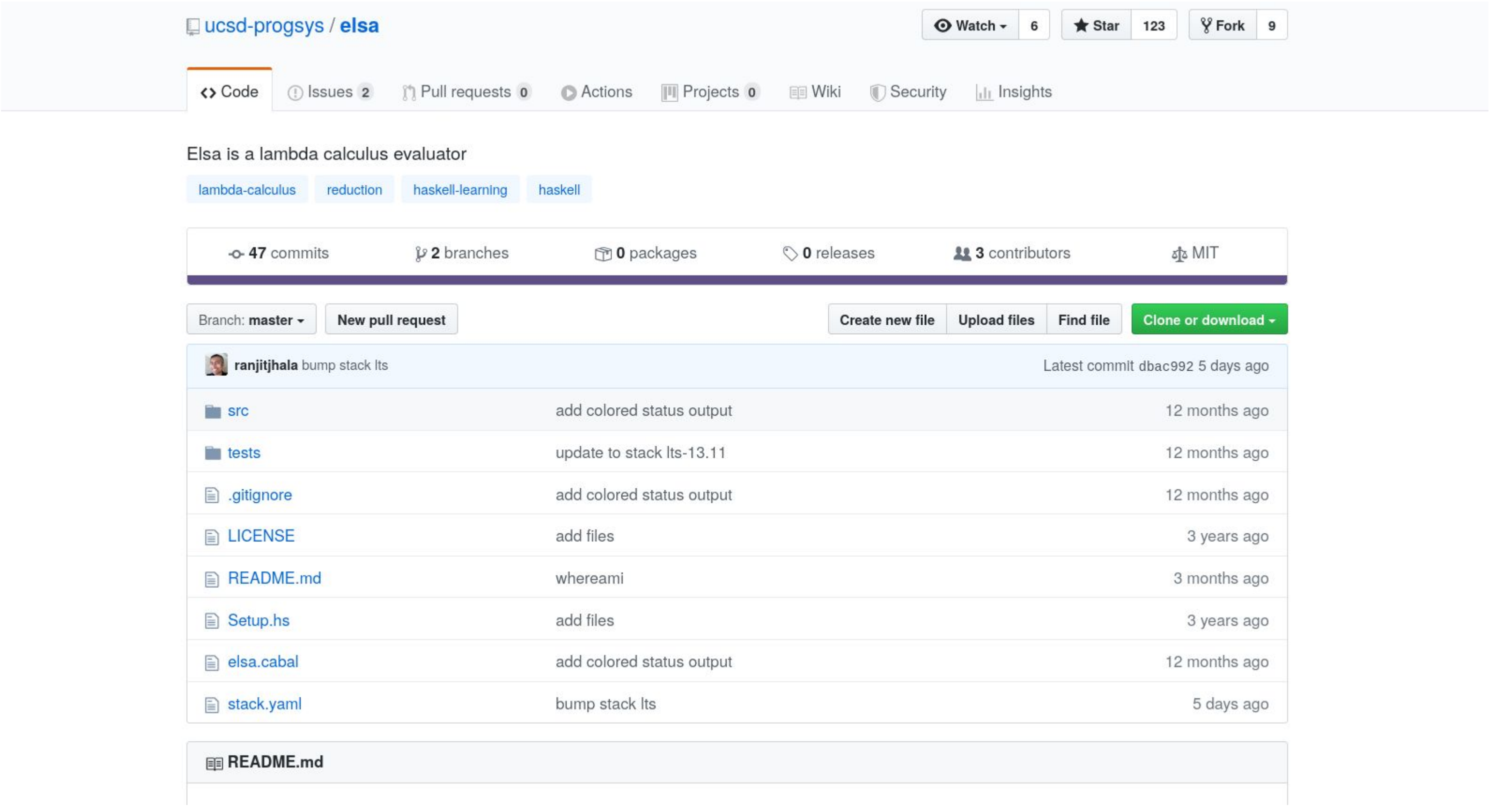
Setup

HW0 will have you *manually* evaluate lambda calculus terms

Elsa checks that **reductions** are valid

Elsa - what is it and how do I use it?

Elsa is implemented as a Haskell package



elsa: A tiny language for understanding the lambda-calculus

[language, library, mit, program] [Propose Tags]

elsa is a small proof checker for verifying sequences of reductions of lambda-calculus terms. The goal is to help students build up intuition about lambda-terms, alpha-equivalence, beta-reduction, and in general, the notion of computation by substitution.

[Skip to Readme]

Modules

[Index] [Quick Jump]

Language

- Language.Elsa
  - Language.Elsa.Eval
  - Language.Elsa.Parser
  - Language.Elsa.Runner
  - Language.Elsa.Types

Downloads

- elsa-0.2.1.2.tar.gz [browse] (Cabal source package)
- Package description (as included in the package)

Maintainer's Corner

For package maintainers and hackage trustees

- edit package information

Versions [faq]

0.1.0.0, 0.1.0.1, 0.2.0.0, 0.2.0.1, 0.2.1.0, 0.2.1.1, **0.2.1.2**

Dependencies

ansi-terminal, array, base (==4.*), dequeue, directory, elsa, filepath, hashable, json, megaparsec (>=7.0.4), mtl, unordered-containers [details]

License

MIT

Author

Ranjit Jhala

Maintainer

jhala@cs.ucsd.edu

Category

Language

Home page

http://github.com/ucsd-progsys/elsa

Source repo

head: git clone https://github.com/ucsd-progsys/elsa/

Uploaded

by ranjitjhala at Mon Apr 1 19:56:43 UTC 2019

Distributions

NixOS:0.2.1.2

Executables

elsa

How do I run elsa and do the HW?

Options:

1. SSH into ieng6
2. Install stack locally
3. Use online demo

SSH into ieng6

Pros:

- Should have everything installed already
- Standardized and easy for us to help us with

Cons:

- As of this morning, ITS had not configured the environment correctly

Install stack locally

Pros:

- Everything can be done offline
- We will use Haskell throughout the class, you might want it locally

Cons:

- Installing stack might be annoying
- Unix: should be easy
- Mac: should also be easy with brew
- Windows: ??
- WSL: Didn’t work last time I tried it, but might work now?

Online demo

Pros:

- Will “just work”

Cons:

- Very clumsy for doing the homework

Doing the homework

`make test` will check your work

`make turnin` or `git push` turns in the homework

Do not use =*> or =~> operators!

Agenda

Setup

What is the lambda calculus

Syntax

Alpha/Beta reductions

PA0 tips

λ

What is the lambda calculus

Very simple programming language

Still Turing complete (???)

What is the lambda calculus

It might look silly but...

- Simple **formal model** of programming
- Provides a minimal framework for exploring and reasoning about various PL concepts
- Fundamental to lots of PL research (especially functional programming)
- **Definitely on the exam**

Agenda

Setup

What is the lambda calculus

Syntax

Alpha/Beta reductions

PA0 tips

λ

Syntax

`x` : Variable

`(\x -> M)` : Function abstraction (M is a lambda term)

`(M N)` : Function application (M, N are lambda terms)

All we can do is declare functions and apply functions!

Functions are *first-class*: We can apply functions to other functions, and a function can return another function

Syntax

```
\a -> (\b -> b)  -- Function that takes a parameter “a” and
                 -- returns a function that takes a param “b”
\a -> \b -> b    -- Syntactic sugar for above
\a b -> b        -- More syntactic sugar
```

Agenda

Setup

What is the lambda calculus

Syntax

Alpha/Beta reductions

PA0 tips

Alpha/Beta reductions

Beta step: Calling a function

Alpha step: Renaming a variable inside a function

Beta step

What do we do with `(\x -> x) y` ?

We can **substitute** y for x inside the body of the function: we just get `y`

More examples:

`(\a b c -> b) d` becomes `(\b c -> b)`

`(\b c -> b) e` becomes `(\c -> e)`

`(\a b c -> b) d e` becomes `(\c -> e)`

In Elsa

```
eval beta :
  (\f x -> f (f x)) g
```

In Elsa

```
eval beta :
  (\f x -> f (f x)) g
  =b> \x ->  g (g x)
```

In Elsa

```

eval beta :
  (\f x -> f (f x)) g
  =b> \x ->  g (x x)

```

What if things get weird?

```
eval beta2 :
  (\x y z -> z y x) y z x -- uh oh
```

Can we still perform a beta reduction?

```
eval beta2 :
  (\x y z -> z y x) y z x
  =b> (\y z -> z y y) z x
```

```

eval beta2 :
  (\x y z -> z y x) y z x
  =b> (\y z -> z y y) z x

```

This doesn’t work!

“y” in the argument is a concrete value

“x” and “y” in the function are purely symbolic -- they just refer to the second argument. So when we substitute “y” for “x”, we are using the same name to refer to two different things. This doesn’t make sense!

We need to be able to rename variables

Alpha steps let you do just this:

```
eval alpha_equiv :
  (\x -> x)
  =a> (\y -> y)
  =a> (\z -> z)
```

We use alpha steps to enable beta steps

```
eval beta2 :
  (\x y z -> z y x) y z x -- uh oh
```

We use alpha steps to enable beta steps

```
eval beta2 :
  (\x y z -> z y x) y z x
  =a> (\x a z -> z a x) y z x
  =b> (\a z -> z a y) z x
```

Agenda

Setup

What is the lambda calculus

Syntax

Alpha/Beta reductions

PA0 tips

PA0 Overview

Goal: Simplify lambda calculus expressions via alpha/beta steps

You will need to understand:

- How to apply alpha/beta steps
- The definitions in each source file

Be aware: the lambda calculus is weird! This might take time

PA0 Overview

Each problem will define higher-level concepts with lambda terms:

```
--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2
```

Most of these definitions will not make sense on their own!

TRUE and FALSE make no sense without the definition of ITE -- you need to read all the definitions and try to figure out how they work together

PA0 overview

Elsa also offers a =d> operator

This allows you to replace symbols with their definition -- this is key! Use it early

```
--------------------------------------------------------------------------------
-- DO NOT MODIFY THIS SEGMENT
--------------------------------------------------------------------------------

let TRUE  = \x y -> x
let FALSE = \x y -> y
let ITE   = \b x y -> b x y
let NOT   = \b x y -> b y x
let AND   = \b1 b2 -> ITE b1 b2 FALSE
let OR    = \b1 b2 -> ITE b1 TRUE b2

--------------------------------------------------------------------------------
-- YOU SHOULD ONLY MODIFY THE TEXT BELOW, JUST THE PARTS MARKED AS COMMENTS
--------------------------------------------------------------------------------

eval not_true :
  NOT TRUE
  -- (a) fill in your reductions here
  =d> FALSE
```

However, you can also make the problems too complicated...

If we replace **all** definitions, we might end up with too much complexity!

Which of these is easier to work with? Why?

```
eval not_true :
  NOT TRUE
  =d> (\b x y -> b y x) TRUE
```

```
eval not_true :
  NOT TRUE
  =d> (\b x y -> b y x) (\x y -> x)
```

Examples