CMSC330: Review and Finale

Chris Kauffman

Last Updated:
Thu Dec 7 09:23:31 AM EST 2023

Logistics

Goals

- Rust Wrap
- Moving Ahead
- Review

Kauffman OH

- Mon 11-Dec: 1-3pm
- Tue 12-Dec: 1-3pm

Previously indicated Tue OH would not happen due to travel but this was incorrect

Date	Event
Tue 05-Dec	Rust-wrap
	Expression Problem
Thu 07-Dec	Review Problems
Fri 08-Dec	Dis: Quiz 4
Mon 11-Dec	Project 8 Due
	Kauffman OH 1-3pm
Tue 12-Dec	Reading Day
	Kauffman OH 1-3pm
Wed 13-Dec	Final Exam
	4-6pm
	ESJ 0224

Beyond CMSC3330

Coursework

CMSC430 Introduction to Compilers Obvious follow-on which studies the construction of compilers and interpreters, often taught in Racket

CMSC433 Programming Language Technologies and Paradigms Topics vary by instructor, usually focuses tailoring of language features to tackle certain problems like concurrency

Self-Study

The most interesting PL's I've seen in the last 10 years are

Clojure A modern Lisp that runs on the Java Virtual Machine specifically targeted at enabling concurrency within programs and allowing coders to solve problems rather than wrestle with the language

Julia Targeted at solving numerical problems and providing a more efficient, nicer experience than Matlab/Octave, has a fascinating type system which is at once dynamic with multiple dispatch but smacks of static features like OCaml's type system

Requested Topics

- Operational Semantics Problem
- Lambda Calculus: alpha conversion
- Lambda Calculus: beta reductions
- OCaml practice problems

Operational Semantics: StringLang Intro

Examples

```
{reverse,"dog"}
{concat,{suffix,2,"4321"},"JumpStreet"}
{suffix, 4, {concat,"abc",{reverse,"123"}}}
```

CFG

```
E -> {reverse,E}
E -> {concat,E,E}
E -> {suffix,N,E}
E -> "string"
N -> natural number {0,1,2,3,....}
```

Operational Semantics: StringLang Evaluation Rules

English and/or Python as a Meta Language

- `N` is a natural number like 0,1,2,...
- `S` is a string like `"dog"` or `"Tachion"`

Evaluation Rules

```
----    ----   SELF EVALUATION
S=>S    N=>N   for strings/nums

          (Python: S2 = S1[-N:])
E1=>S1    S2 is last N chars of S1
---------------------------------  SUFFIX
      {suffix, N, E1} => S2

               (Python: S3=S1+S2)
E1=>S1  E2=>S2  S3 is "S1..S2"
-----------------------------    CONCAT
      {concat, E1, E2}=>S3

        (Python: S2=S1[::-1])
E1=>S1  S2 is S1 in reverse order
---------------------------------  REVERSE
         {reverse, E1}=>S
```

Sample Evaluations

```
{concat, "tri", "gun"}
=> "trigun"

{reverse, "god"}
=> "dog"

{suffix, 3, "poignant"}
=> "ant"

{concat, {suffix, 2, "hippo"}, "pular"}
=> "popular"

{concat, {reverse, {suffix, 3, "pokemon"}},
         "nom"}
=> "nomnom"
```

Exercise: Operational Semantics: StringLang Derivation

```
??????????
```

```
=============================================================================
    {concat, {reverse, {suffix, 3, "pokemon"}}, "nom"} => "nomnom"
```

Answers: Operational Semantics: StringLang Derivation

```
=========
"pokemon"  "mon"="pokemon"[-3:]
==============================
{suffix, 3, "pokemon"}=>"mon"  nom="mon"[::-1]
=============================================    =====
{reverse, {suffix, 3, "pokemon"}} => "nom"        "nom"  "nomnom"="nom"+"nom"
=========================================================================
     {concat, {reverse, {suffix, 3, "pokemon"}}, "nom"} => "nomnom"
```

Lambda Calculus: Alpha Conversion

- Two terms that differ only in the names bound variables are considered identical if they only differ in the names of bound variables
- Examples:
  1. $\lambda x.x \equiv \lambda y.y$ *(Both Identity Function)*

```
L x.x = L y.y
```

  1. $\lambda x.\lambda y.x\, y \equiv \lambda q.\lambda r.q\, r$

```
L x.L y.  x y = L q.  L r.  q r
```

  1. $\lambda a.\lambda b.z\, a \equiv \lambda t.\lambda u.z\, a$

```
L a.L b.  z a = L t.  L u.  z t
```

- Lingo: "Terms are equal up to renaming of bound variables."
- Church dubbed this "Alpha-Conversion": consistently rename bound variables to reveal structural equivalence
- NOTE: **free variables** do not get renamed during Alpha Conversion

Lambda Calculus: Alpha Conversion Exercise

In the following us variable names `v0, v1, v2...` when Alpha Converting.

1. Alpha Convert

   ```
   L a.L b.z a
   L t.L u.z t
   ```

   to show that they are equivalent.

2.Alpha Convert the following

```
L x.L y.L x.L y.y x =alpha=> ??
```

3.Determine if the following two terms are "Equal up to renaming of bound variables"

```
Equal?
(L t.L u.(t a) u) (L t.t (L a.t))
(L x.L z.(x a) z) (L a.a (L z.a))
```

Lambda Calculus: Alpha Conversion **Answers**

In the following us variable names `0, 1, 2...` when Alpha Converting.

1. Alpha Convert

   ```
   L a.L b.z a =alpha=> L 0.L 1.z 0
   L t.L u.z t =alpha=> L 0.L 1.z 0
   ```

   to show that they are equivalent.

2. Alpha Convert the following

   ```
   L x.L y.L x.L y.y x =alpha=> L 0.L 1.L 2.L 3.3 2
   ```

3. Determine if the following two terms are "Equal up to renaming of bound variables"

   ```
   (L t.L u.(t a) u)  (L t.t (L a.t)) =alpha\
   (L 0.L 1.(0 a) 1)  (L 2.2 (L 3.2)) <------+
   (L 0.L 1.(0 a) 1)  (L 2.2 (L 3.2)) <------+
   (L x.L z.(x a) z)  (L a.a (L z.a)) =alpha/
   ```

   YES equivalent

Lambda Calculus: Beta Reduction

- Lambda Calculus has only 1 operation: *Apply a Function (Abstraction)*
- Two adjacent terms indicate an Application
- `t1 t2` is `t1` Applied to `t2`
- If `t1` is an abstraction of `L v. body` the application may be **reduced** by
  1. Removing the `L v` and `t2`
  2. Replacing all occurrences `v` in `body` with `t2`
- This is a single **reduction step**
- In **Full Beta Reduction** terms are reduced until the no further reductions are possible at which point the term is in **Beta Normal Form**

Exercise: Evaluation Strategies

- While evaluating a Lambda calculus term, there may be several choices of function applications to reduce
- An **Evaluation Strategy** dictates which term to reduce; we discussed 2 common Evaluation Strategies
  1. Call by Name (Lazy Eval)
  2. Call by Value (Eager Eval)

Questions to conisder

1. What's the difference between these two evaluation strategies?
2. When reducing a lambda term to Beta Normal Form, does it matter which evaluation strategy / order is used?

Answers: Evaluation Strategies

Questions to consider:

1. What's the difference between these two evaluation strategies?
   - Call by Name / Lazy Eval: Reduce the leftmost, outermost term first.
   - Call by Name / Lazy Eval: If the term is an Application, reduce the right-hand side first. Then reduce the leftmost, outermost term first.

   Will want examples of these for concreteness.
2. When reducing a lambda term to Beta Normal Form, does it matter which evaluation strategy / order is used?
   It's comlicated: In most cases, evaluation order won't matter. However, if the term "loops/recurses" through certain *combinators*, some orders of evaluation may diverge infinitely. The Church-Rossner Theorem proves that if all sequences of reductions that converges to a normal form converge to the same normal form.

Exercise: Beta Reduce by Eval Strategy

Show each reduction step on the terms below using both Lazy and Eager Evaluation Strategies. Determine if the ending terms are the same.

LAZY 1

```
(L z.z) ((L y.y) x)
```

EAGER 1

```
(L z.z) ((L y.y) x)
```

LAZY 2

```
(L x.(L y.y) ((L z.z) (x x)))
```

EAGER 2

```
(L x.(L y.y) ((L z.z) (x x)))
```

Answers: Beta Reduce by Eval Strategy

Show each reduction step on the terms below using both Lazy and Eager Evaluation Strategies. Determine if the ending terms are the same.

```
LAZY 1
   (L z.z) ((L y.y) x)
      ^    ^^^^^^^^^^

=> (L y.y) x
      ^    ^

=> x
```

```
EAGER 1
   (L z.z) ((L y.y) x)
                ^    ^

=> (L z.z) x
      ^    ^

=> x
```

```
LAZY 2
   (L x.(L y.y) ((L z.z) (x x)))
            ^    ^^^^^^^^^^^^^^

=> (L x.((L z.z) (x x)))
             ^    ^^^^^

=> (L x.(x x))
```

```
EAGER 2
   (L x.(L y.y) ((L z.z) (x x)))
                         ^    ^^^^^

=> (L x.(L y.y) (x x))
             ^    ^^^^^

=> (L x.(x x))
```

Yes : terms reduce to the same Beta Normal Form for both Lazy/Eager strategies

Reminders on Diverging Definitions

`(L x.(L y.y) ((L z.z) (x x))) - Value or Reducable?`

- This term is interesting as it is a "Function" (abstraction)
- The definitions for Lazy/Eager evaluation in CMSC330 is to perform reductions "inside" the function
- I have observed that reliable, published sources discussing Lambda Calculus **would not** reduce this term according to their definitions of the same evaluation strategies
- I'll be working to adjust the culture of the class to favor definitions used by reliable, published sources of information on the subject
- But, if faced with an exam question on this, reduce away...

Additional Reduction Examples to consider

Try these with Lazy/Eager evaluation strategies for additional practice / discussion.

1. `(L a.b a) ((L x.y z))`

2. `(x (L y. y)) ((L z.z) a)`

3. `(x ((L y. y) b)) ((L z.z) a)`

3 is interesting. . .

Exercise: A Previous Rust Problem

```
1  fn make_vec() -> Vec<&String>{
2    let s = String::from("abc");
3    let mut v = vec![];
4    v.push(&s);
5    return v;
6  }
7  fn use_make_vec() {
8    let v = make_vec();
9    println!("v[0]: {}",v[0]);
10 }
```

Compiling the nearby Rust code yields an error `rustc` suggests the problem may be resolved by changing the first function prototype to:

```
fn make_vec() -> Vec<&'static String>{
```

1. What is the essence of the problem?
2. Will the suggested change fix the code?

Answers: A Previous Rust Problem

```
1  fn make_vec() -> Vec<&String>{
2    let s = String::from("abc");
3    let mut v = vec![];
4    v.push(&s);
5    return v;
6  }
7  fn use_make_vec() {
8    let v = make_vec();
9    println!("v[0]: {}",v[0]);
10 }
```

Compiling the nearby Rust code yields an error `rustc` suggests the problem may be resolved by changing the first function prototype to:

```
fn make_vec() -> Vec<&'static String>{
```

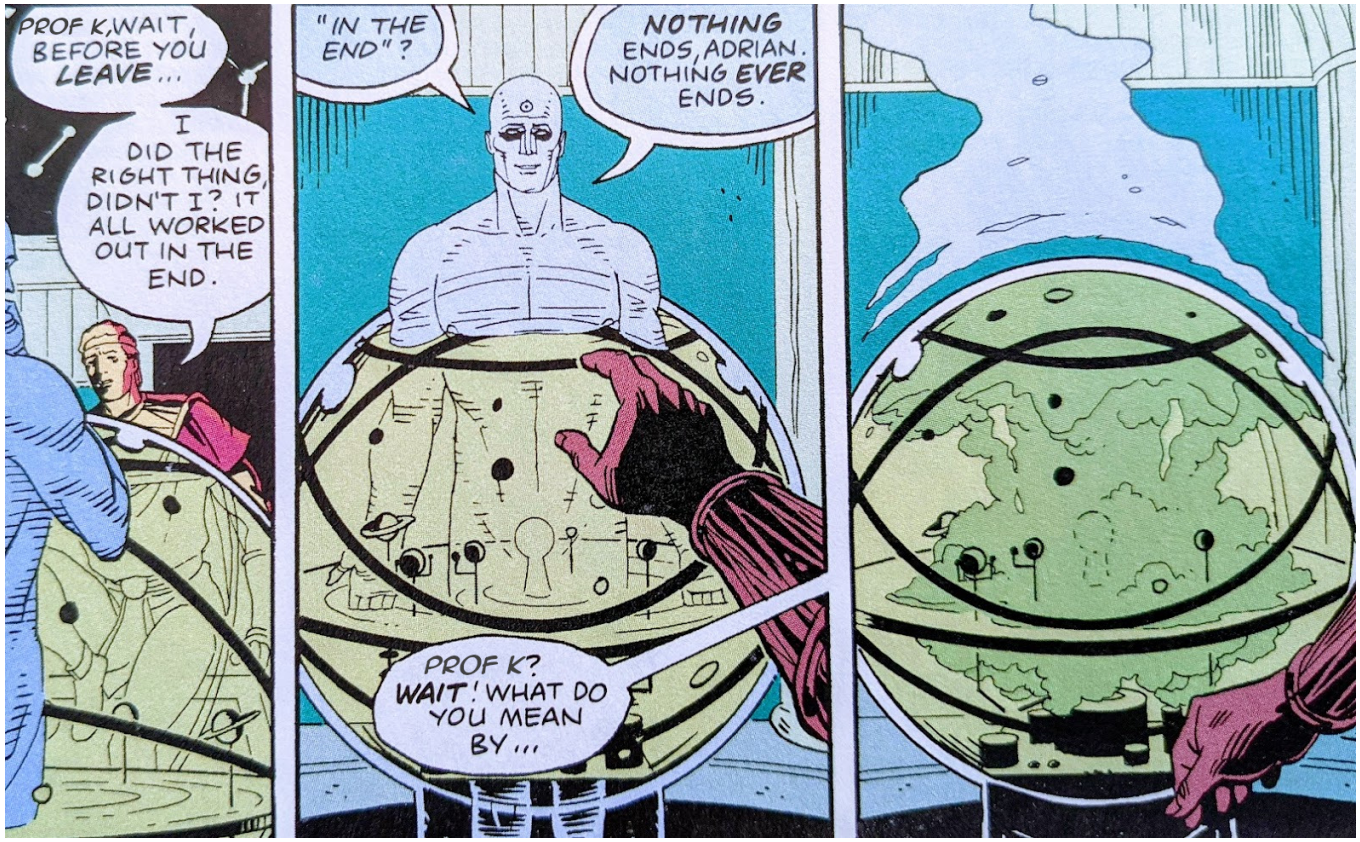
1. What is the essence of the problem?

   `make_vec()` allocates a Vector and a String, adds a Reference to the String into the Vector, then attempts to return the Vector. The String will be dropped leading to the Vector containing a dangling reference. `rustc` sees this and call a foul.

2. Will the suggested change fix the code?

   No: if the prototype is changed, `rustc` will instead complain that the lifetime of the String ref is not the same as the return type. The fundamental problem is that this is dangerous so Rust won't allow it. Adding the String directly (not as a ref) would cause the Vector to own the String data leading to success with mild changes of types.

Nothing Ever Ends



- What you learned will recur in your career at some point and demonstrate whether you learned it well the first time or need another pass.
- Some of it will change in the future and make you feel old.
- Expect this and stay determined.

Conclusion

It's been a hell of a semester.
I'm proud of all of you.
Keep up the good work.
Stay safe. Happy Hacking.