CPSC-354 Report

Maxwell Rovenger
Chapman University

November 10, 2024

Abstract

Contents

1 Introduction

(Delete and Replace): This report will document your learning throughout the course. It will be a collection of your notes, homework solutions, and critical reflections on the content of the course. Something in between a semester-long take home exam and your own lecture notes.[1]

To modify this template you need to modify the source `report.tex` which is available in the course repo. For guidance on how to do this read both the source and the pdf of `latex-example.tex` which is also available in the repo. Also check out the usual resources (Google, Stackoverflow, LLM, etc). It was never as easy as now to learn a new programming lanugage (which, btw, LaTeX is).

[1] One purpose of giving the report the form of lecture notes is that self-explanation is a technique proven to help with learning, see Chapter 6 of Craig Barton, How I Wish I'd Taught Maths, and references therein. In fact, the report can lead you from self-explanation (which is what you do for the weekly deadline) to explaining to others (which is what you do for the final submission). Another purpose is to help those of you who want to go on to graduate school to develop some basic writing skills. A report that you could proudly add to your application to graduate school (or a job application in industry) would give you full points.

For writing LaTeX with VSCode use the LaTeX Workshop extension.

There will be deadlines during the semester, graded mostly for completeness. That means that you will get the points if you submit in time and are on the right track, independently of whether the solutions are technically correct. You will have the opportunity to revise your work for the final submission of the full report.

The full report is due at the end of the finals week. It will be graded according to the following guidelines.

2 Week by Week

2.1 Week 1

Notes

Lectures introduced Lean, a programming language to help prove discrete mathematics proofs. Through using the Natural Numbers Game, we saw how Lean operates and how it works it can, with functions acting as steps, prove theorems similar to how we did in discrete math with pen and paper and induction.

Homework

Solved problems in Natural Numbers Game using Lean to help recap teachings from Discrete Mathematics. Specifically dealing with successors and predecessors and how they can be used to change certain sides to equal the other.

Level 5/8:

```
a + (b + 0) + (c + 0) = a + b + c
rw [add_zero]
a + b + (c + 0) = a + b + c
rw [add_zero]
a + b + c = a + b + c
rfl
```

Level 6/8:

```
a + (b + 0) + (c + 0) = a + b + c
rw[add_zero c]
a + (b + 0) + c = a + b + c
rw[add_zero b]
a + b + c = a + b + c
rfl
```

Level 7/8:

```
succ n = n + 1
rw[one_eq_succ_zero]
succ n = n + succ 0
rw[add_succ]
succ n = succ (n + 0)
rw[add_zero]
succ n = succ n
rfl
```

Level 8/8:

```
2 + 2 = 4
rw[four_eq_succ_three]
2 + 2 = succ 3
rw[three_eq_succ_two]
2 + 2 = succ (succ 2)
rw[two_eq_succ_one]
succ 1 + succ 1 = succ (succ (succ 1))
rw[add_succ]
succ (succ 1 + 1) = succ (succ (succ 1))
rw[succ_eq_add_one]
succ 1 + 1 + 1 = succ (succ (succ 1))
rw[succ_eq_add_one]
1 + 1 + 1 + 1 = succ (succ (1 + 1))
rw[succ_eq_add_one]
1 + 1 + 1 + 1 = succ (1 + 1) + 1
rw[succ_eq_add_one]
1 + 1 + 1 + 1 = 1 + 1 + 1 + 1
rfl
```

For level 5/8 specifically, we can see that the Lean proof, `rw[add_zero]`, corresponds to Proof Algorithm 1: Addition, in that any variable added to zero will ultimately equal just the variable. For example: $a + 0 = a$.

From this homework, I learned how to use the Lean proof and saw how each function operated exactly like algorithms and proofs I had used in Discrete Mathematics.

Comments and Questions

Although this week generally served as just an introduction to the curriculum and a recap of discrete mathematics, I am curious as to whether or not there was a better example to be shown of how exactly a computer uses discrete mathematics. At this point, I am knowledgeable of how discrete mathematics operates and how computers can use operators to conduct mathematical operations, but I have yet to see a direct example of a computer "thinking" through a math calculation.

If the way computers have been taught mathematics is based purely on successors and predecessors, does that make other operations like multiplication, division, and exponents far more taxing on a CPU since they have to calculate an incredible amount of successors, or do the operators used to cause multiplication and exponentiation ignore that by just creating duplicates and adding them together?

2.2 Week 2

Notes

Recursion, in coding, allows for more simplistic code that is easier to read, scale, and apply.

Regarding the allowing of typos in coding, code should be non-ambigious because coding should be universal and should work the same within the same virtual machine regardless of how you access it. We see examples of this in GitHub's copilot since it only makes suggestions that are non-ambigious and can work on any machine, as long as they have the correct virtual machine.

Lean contains tactics, Ex: rw, and theorems, Ex: one_eq_succ_zero. Tactics are commands while theorems are logical propositions.

Homework

Level 1/5:

```
0 + n = n
induction n with d hd
0 + 0 = 0
rw[add_zero]
0 = 0
rfl
0 + succ d = succ d
rw[add_succ]
succ (0 + d) = succ d
rw[hd]
succ d = succ d
rfl
```

Level 2/5:

```
succ a + b = succ (a + b)
induction b with b
succ a + 0 = succ (a + 0)
rw[add_zero]
succ a = succ (a + 0)
rw[add_zero]
succ a = succ a
rfl
succ a + succ b = succ (a + succ b)
rw[add_succ]
succ (succ a + b) = succ (a + succ b)
rw[n_ih]
succ (succ (a + b)) = succ (a + succ b)
rw[add_succ]
succ (succ (a + b)) = succ (succ (a + b))
rfl
```

Level 3/5:

```
a + b = b + a
induction b with b hb
a + 0 = 0 + a
rw [add_zero]
a = 0 + a
rw [zero_add]
a = a
rfl
a + succ b = succ b + a
rw [add_succ]
succ (a + b) = succ b + a
rw [hb]
succ (b + a) = succ b + a
rw [succ_add]
succ (b + a) = succ (b + a)
rfl
```

Level 4/5:

```
a + b + c = a + (b + c)
induction c with c hc
a + b + 0 = a + (b + 0)
rw [add_zero]
a + b = a + (b + 0)
rw [add_zero]
a + b = a + b
rfl
a + b + succ c = a + (b + succ c)
rw [add_succ]
succ (a + b + c) = a + (b + succ c)
rw [add_succ]
succ (a + b + c) = a + succ (b + c)
rw [add_succ]
succ (a + b + c) = succ (a + (b + c))
rw [hc]
succ (a + (b + c)) = succ (a + (b + c))
rfl
```

Level 5/5:

```
a + b + c = a + c + b
induction c with c hc
a + b + 0 = a + 0 + b
rw [add_zero, add_zero]
a + b = a + b
rfl
a + b + succ c = a + succ c + b
rw [add_succ]
succ (a + b + c) = a + succ c + b
rw [add_succ]
succ (a + b + c) = succ (a + c) + b
rw [succ_add]
succ (a + b + c) = succ (a + c + b)
rw [hc]
succ (a + c + b) = succ (a + c + b)
rfl
```

Comments and Questions

Personally, I felt that when it comes to AI autocorrecting code, instead of being lenient towards mistyped code, AI could just simply correct the typos. This would allow for a smoother coding process wihle also creating non-ambigious code.

After playing around with the Tower of Hanoi, it is apparent that recursion is invaluable in programming. We often see it as a way to neatly write code but it can also be massive time saver as apparent with the Tower of Hanoi where instead of having to, through trial and error finding out the solution, we can just use a recursive equation and plug in our values. My question then is, in what other fields can we apply recursion, as it is becoming more apparent to me that recursion can be a form of thinking rather than a field specific method?

Lean Proof

$$\begin{aligned} a+b+c &= a+(b+c) \\ a+b+0 &= a+(b+0) == defof+ \\ a+b &= a+(b+0) == defof+ \\ a+b &= a+b \end{aligned}$$

2.3 Week 3

Notes

It is impossible for a computer to prove all mathematical proofs since there are statements in arithmetic that are unclear if they are true or not and thus cannot be proved. This is due to there being proofs in mathematics that are built based upon assumption and although there are no problems found so far, they cannot be proved through ordinary means like what the computer would opt to do.

Homework

LLM Literature Review README

Discord Posting

For my literature review, I used ChatGPT4 to ask questions about recursion within computer systems. More specifically, if it decreases memory usage or provides any processing speed based benefits rather than just readability or integration based benefits. Through my conversation, I learned that due to creating more and more stack frames for each iteration, recursion can actually increase memory usage. However, there is an alternative, that being tail recursion. It was developed in 1970s however it either does not have support or only has limited support for popular languages like Python and JavaScript. Thus, it was concluded that recursive functions are actually less memory efficient than iterative solutions. This surprised me as through recursion, humans are able to abstract a lot of our thinking and are able to quickly find out what the next iterative output is. This contrasts computers which require more processing power to perform recursive methods. This creates an inverse relationship where the better the human understanding, the worse the computer understanding.

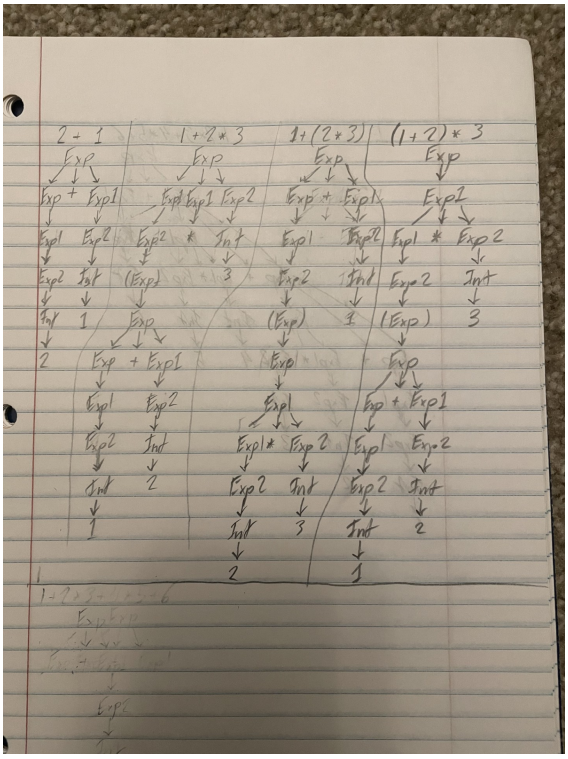
2.4 Week 4

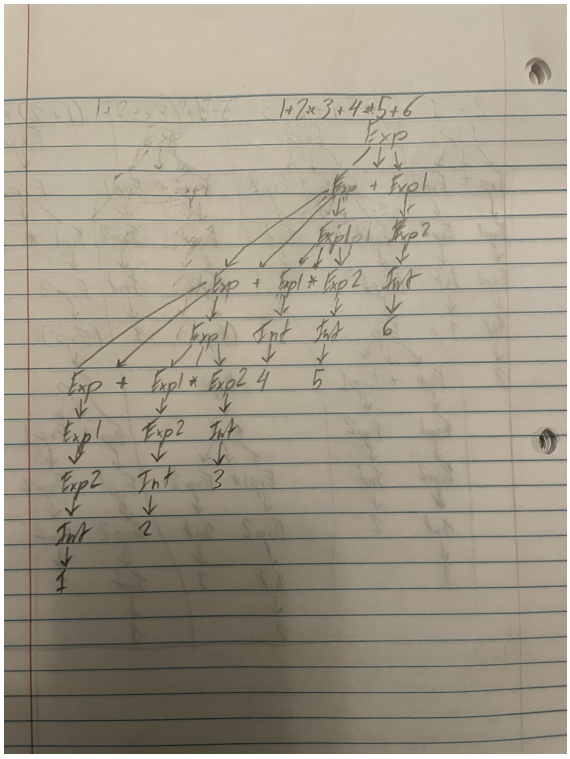
Notes

An abstract syntax tree is the result of parsing and helps display the order and relationship between operators in a mathematical problem. the abstract syntax tree is the most intermediate, as it is more drawn out than both concrete syntax and a concrete syntax tree.

We can define well formed expressions through formal grammar. Since formal grammar helps to create definitions for what becomes a well-formed expression.

Homework





Questions and Comments

If my understanding is correct, all formal languages are allowed to have their own alphabets and grammar that must be defined. However, we have found ourselves in a position where the majority of, if not all, programming langauges will use the same characters or symbols for operators and values. Thus, my question is if there is a protocol enforced that mandates the formal language used in a programming languages or has this all occurred just for ease of use?

2.5 Week 5

Notes

In this week's lecture, we learned about the logic behind AND and how computer's process it. This shows that a proof in logic can be represented as a program in a programming language and vice versa.

Similarly, in the natural numbers game, we saw the application of these theorems which allowed us to prove theorems in Lean. So if a refresher is needed, going through the "A Lean Intro To Logic" is a good idea.

Homework

Level 1/8:

```
todo_list: P
exact todo_list
```

Level 2/8:

$[Goal : P \wedge S]$
```
exact and_intro p s
```

Level 3/8:

$[Goal : (A \wedge I) \wedge O \wedge U]$
```
exact and_intro (and_intro a i) (and_intro o u)
```

Level 4/8:

```
Assumptions:
```
$[-vm : P \wedge S]$
```
have p := vm.left
Assumptions:
```
$[-vm : P \wedge S]$
$[-p : P]$
```
exact p
```

Level 5/8:

```
Assumptions:
```
$[-h : P \wedge Q]$
```
have q := and_right h
Assumptions:
```
$[-h : P \wedge Q]$
$[-q : Q]$
```
exact q
```

Level 6/8:

```
Assumptions:
```
$[-h1 : A \wedge I]$
$[-h2 : O \wedge U]$
```
have A := and_left h1
Assumptions:
```
$[-h1 : A \wedge I]$
$[-h2 : O \wedge U]$
$[-A : A]$
```
have U := and_right h2
Assumptions:
```
$[-h1 : A \wedge I]$
$[-h2 : O \wedge U]$
$[-A : A]$
$[-U : U]$
```
exact and_intro A U
```

Level 7/8:

```
Assumptions:
```
$[-h : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L]$
```
have h1 := and_left h
Assumptions:
```

$[-h : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L]$
$[-h1 : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$

```
have h2 := and_right h1
```

Assumptions:

$[-h : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L]$
$[-h1 : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h2 : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$

```
have h3 := and_left h2
```

Assumptions:

$[-h : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L]$
$[-h1 : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h2 : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h3 : (L \wedge C) \wedge L]$

```
have h4 := and_left h3
```

Assumptions:

$[-h : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L]$
$[-h1 : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h2 : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h3 : (L \wedge C) \wedge L]$
$[-h4 : L \wedge C]$

```
have h5 := and_right h4
```

Assumptions:

$[-h : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L]$
$[-h1 : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h2 : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L]$
$[-h3 : (L \wedge C) \wedge L]$
$[-h4 : L \wedge C]$
$[-h5 : C]$

```
exact h5
```

Level 8/8:

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$ `and_left (1)`

```
have h1 := and_left h
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$

```
have h2 := and_left h1
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$

```
have h3 := and_right h1
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$

```
have h4 := and_right h
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$

$[-h4 : \neg I \wedge (C \wedge \neg O) \wedge \neg U]$

```
have h5 := and_right h4
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$
$[-h4 : \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h5 : (C \wedge \neg O) \wedge \neg U]$

```
have h6 := and_left h5
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$
$[-h4 : \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h5 : (C \wedge \neg O) \wedge \neg U]$
$[-h6 : C \wedge \neg O]$

```
have h7 := and_left h6
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$
$[-h4 : \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h5 : (C \wedge \neg O) \wedge \neg U]$
$[-h6 : C \wedge \neg O]$
$[-h7 : C]$

```
have h8 := and_left h2
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$
$[-h4 : \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h5 : (C \wedge \neg O) \wedge \neg U]$
$[-h6 : C \wedge \neg O]$
$[-h7 : C]$
$[-h8 : P]$

```
have h9 := and_right h2
```

Assumptions:

$[-h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h1 : (P \wedge S) \wedge A]$
$[-h2 : P \wedge S]$
$[-h3 : A]$
$[-h4 : \neg I \wedge (C \wedge \neg O) \wedge \neg U]$
$[-h5 : (C \wedge \neg O) \wedge \neg U]$
$[-h6 : C \wedge \neg O]$
$[-h7 : C]$
$[-h8 : P]$
$[-h9 : S]$

```
exact and_intro h3 (and_intro h7 (and_intro h8 h9))
```

---

level 8/8 in Mathematical Logic

(1)	$((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$	assumption
(2)	$(P \wedge S) \wedge A$	and_left (1)
(3)	$P \wedge S$	and_left (2)
(4)	A	and_right (2)
(5)	$\neg I \wedge (C \wedge \neg O) \wedge \neg U$	and_right (1)
(6)	$(C \wedge \neg O) \wedge \neg U$	and_right (5)
(7)	$C \wedge \neg O$	and_left (6)
(8)	C	and_left (7)
(9)	P	and_left (3)
(10)	S	and_right (3)

Comments and Questions

Following our exposure to proving the new theorem regarding "AND" in Lean, I took note of the recursive nature of this course. Specifically, how all topics in this course build up on each other. This lead me to asking the question: How does recursion play into isomorphism and is the scalability of recursive functions a necessity for theorems and logic within programming?

2.6 Week 6

Notes

Implication is actually just a function that has to be proved through theorems and proofs rather than just visual confirmation.

Precedence: An integral part of programming languages, decides the order of operations in a mathematical problem. This is important as it can change the outcome of a problem if not done correctly.

- Appraisal: Processed left-to-right
- λ Abstraction: Processed right-to-left

Homework

Level 1/9:

`Assumptions:`
$-bakery\_service : P \to C$
`exact bakery_service p`

Level 2/9:

$haveh_1 : C \to C := func : C => c$
`Assumptions:`
$-h_1 : C \to C$
`exact h_1`

Level 3/9:

```
Objects:
  - I S : Prop
```

Goal:
$-I \wedge S \to S \wedge I$
$exact \lambda h : I \wedge S \to and_intro(and\_righth)h.left$

Level 4/9:

Assumptions:
$-h1 : C \to A$
$-h2 : A \to S$
$exact h1 \gg h2$

Level 5/9:

Assumptions:
-p : P
$-h1 : P \to Q$
$-h2 : Q \to R$
$-h3 : Q \to T$
$-h4 : S \to T$
$-h5 : T \to U$
$exact(h1 \gg h3 \gg h5)p$

Level 6/9:

Assumptions:
$-h : C \wedge D \to S$
$exact func => fund => h\langle c, d\rangle$

Level 7/9:

Assumptions:
$-h : C \to D \to S$
$exact fun\langle c, d\rangle => hcd$

Level 8/9:

Assumptions:
$-(S \to C) \wedge (S \to D)$
$exact funs => \langle h.1s, h.2s\rangle$

Level 9/9:

Objects:
- R S : Prop
Goal:
$-R \to (S \to R) \wedge (\neg S \to R)$
$exact funr => \langle fun => r, fun => r\rangle$

Comments and Questions

Although closing laptops allows for better lecture retention, it heavily damages the ability to take notes, decreasing the value of the report's "notes" subsection.

In class, we commonly hear the term "context-free grammar", is it not the case where all grammar within computers should be context free considering it is the foundation for mathematics which always operates the same? What are some examples of grammar that is context specific?

2.7 Week 7

Notes

λ-calc: Useful helpers

1. Projects:
   - (1a) Take 2 args & return 1st lambax.lambday.x (lambdax.lambday.y)
   - (2a)
2. Erase: Take 3 args & return 1st & 3rd: lambdax.lambday.lambdaz.xz
3. Duplicate: take 1 arg & duplicate lambdax.xx

How to encode logic in λ-calc:

1. Specifications:
   - if-then-else true M N = M
   - if-then-else false M N = N
   - true := λx.λy.x
   - false := λx.λy.y

2. Not:
   - not true = false
   - not false = true
   - not M = if-then-else M false true
   - not := λx.x false true

Homework

Comments and Questions

Lambda calculus seems to be extremely low level and be the defining foundation for a lot of programming calculations. With this said, how is it not its own programming language that is in use today for the sake of the fastest possible runtime, sort of similar to how assembly is the bare bones equivalent of programming.

2.8 Week 8/9

Notes

Extensions of lambda calculus
- Basic logic: boolean logic, if-construct
- Basic arithmetic (numerals, add, exp,...)
Want:
- let l = local (names)s
- recursion

Introduce let: Example from VSCode:
- let plus = $\lambda m.\lambda n.xf.\lambda x.mf(ufx)$
in let two = $\lambda f.\lambda x.f(fx)$
in let one = $\lambda f.\lambda x.fx$
in plus one two

Concrete syntax: let x = e, in ez
Goal: define this λ-calc (Church encoding) but can also add this as syntectic sugar.
add to CFG: $c \to c \to$ let id = e in e
abstract syntax:

Y-combinator

$Y := fix := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Claim: (fix F) is an fp of F, ie, F $\to$ F (fix F).
Proof: fixF $\to (\lambda x.F(xx))(\lambda x.F(xx))$

Homework

Exercise 2:
- a b c d $\to$ (((a b) c) d)
- This is because the grammar identifies that there are no parentheses and thus seeks to establish the order of operations.
- (a) $\to$ a
- This is because the grammar identifies that there are parentheses and then seeks to simplify the equation.

Exercise 3:
- Capture-avoiding substitution works by ensuring that when substituting a variable x in an expression, no free variables get mistakenly bound by a new scope. If a substitution would result in a name conflict (i.e., x being replaced in a context where it is already bound), an α-conversion (renaming of bound variables) is performed to avoid variable capture.

```
assert substitute(('lam', 'x', ('lam', 'y', ('var', 'x'), ('var', 'y'))), 'y', ('var', 'x')) ==
    ('lam', 'x', ('lam', 'Var1', ('var', 'x'), ('var', 'Var1')))
print(f"SUBST {MAGENTA}(\\x.\\y.x y) [y/x]{RESET} == ('lam', 'x', ('lam', 'Var1', ('var', 'x'),
    ('var', 'Var1')))")

print("\nCapture-Avoiding Substitution: Test passed!\n")

print("\nevaluate(): All tests passed!\n")
```

- Implemented by:
1. Base case: If the term is a variable and matches the one being substituted, replace it with the expression.
2. Lambda abstraction: When encountering a lambda abstraction $\lambda y.e$, check if the bound variable y conflicts with x . If so, rename y to a fresh variable before substituting.
3. Recursion: Perform the substitution recursively in the body of the lambda or function application, applying α-conversion where necessary to avoid capture.

Exercise 4: - When dealing with capture avoiding substitution, the expected result is not always returned. This is caused by events where there is a scope change that is unanticipated which results in an unexpected outcome. - Not all computations return to normal form since there are computations that have forms where no reductions can occur.

Exercise 5: - The smallest λ-expression that does not reduce to normal form is an omega combinator.
- $\omega = \lambda x.xx$

Exercise 6: - Used debugger

Exercise 7:

- ((m.n. mn) (f.x. f(fx))) (f.x. f(f(fx)))
- Gets interpreted to
- (Var5.((f.(x.(f (f (f x))))) ((f.(x.(f (f (f x))))) Var5)))

Exercise 8:

```
12: eval (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f x))))
    39: eval ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
        25: eval (\f.(\x.(f (f x))))
        19: eval (\f.(\x.(f x)))
    51: substitute ((\m.(\n.(m n))), (\f.(\x.(f (f x)))), m)
        45: eval ((\f.(\x.(f (f x)))) (\f.(\x.(f x))))
            29: eval (\f.(\x.(f x)))
            55: substitute ((\f.(\x.(f (f x)))), (\f.(\x.(f x))), f)
                38: eval (\x.(f (f x)))
                    40: substitute ((\x.(f (f x))), x, x)
                        22: eval (f (f x))
                            14: eval (f x)
```

Comments and Questions

Throughout the course, we've explored countless algorithms and calculations that work recursively. What makes it so difficult to revert any expression, that has been evaluated using a recursive algorithm, back to its original state?

In the course, we've seen that lambda expressions can be reduced into normal form, what exactly is the benefit of having theses expressions displayed in normal form?

2.9 Week 10

Notes

2n $\rightarrow$ n/2
2n+1 $\rightarrow$ 3n+1

c(n) :=
- n/2 n even
- 3n+1 n odd

Conjecture: stabilizes at 1, no matter the initial n.

Rewriting:

Purpose: Formalism, to capture and analyze the process of transforming syntax (aka strings).

Examples:

- High school algebra

- CFGs
- Turing machines
- D/NFA
- MIU save
- Cellular Automata

Definition: an abstract rewrite (or reduction) system ARS is a pair (A, R) with:

- A: set of expressions
- R: set of rewrite rules
- R ¡= A x A a binary relation on A

Computational behavior:
- Confluence and Termination

Homework

1) What did you find most challenging when working through Homework 8/9 and Assignment 3?

When working on Homework 8/9 and Assignment 3, I found it very challenging to both understand and make adjustments to the existing repo which we used for the evaluations. It slowed down my progress by a considerable amount as I had to spend a lot of time understanding the codebase before I could make any changes to it. After finally gaining understanding, I was able to make progress with the exercises and try out the debugger.

2) How did you come up with the key insight for Assignment 3?

I came up with the key insight for Assignment 3 by first understanding the problem and then breaking it down into smaller parts. This process was also enforced by the debugger as it provides the same style of problems solving where we can transform the program into bits and pieces by using the debugger's break points.

3) What is your most interesting takeaway from Homework 8/9 and Assignment 3?

The most interesting takeaway from both assignments, personally, was the fact that the smallest lambda expression that does not reduce to normal form is an omega combinator. This was interesting to me as it showed that not all computations can return to normal form which further supports the format in computation where there are exceptions for almost all rules.

Comments and Questions

Although Langton's Ant has an emergent behavior, is it not the case where when we can visually confirm that the behavior occurs everytime, are we not able to just test the extremes and then be able to verify that this emergent behavior is just its normal behavior and true in all scenarios?

2.10 Week 11

Notes

(2) Terminating if: does ntot admit an infinite computation (a1,a2,a3,...) also called; strictly normalizing.

(3) (weakly) normalizing if every element has some normal form.

Application: Termination analysis of programs → undecidable in general, but can analyze/classify using rewriting theory.

Confluent: If there is no diamond shape in the diagram, it is not confluent.

Unique NF: If we ask if there is a unique normal form, we're asking if all elements have at least one unique normal form.

Properties of ARS: Termination, Normalization, Confluence

ARS: $ba \to ab$
- Is it terminating? (No infinite set)

We can tell that it is terminating if there is a decrease of length in the generating rules.

Not every computation terminates: (a,a,a,a,...)

Termination does imply noramlization.

Homework

1. $A = \emptyset$ **and** $R = \emptyset$

- **Terminating**: Yes, trivially terminating because there are no elements or relations.
- **Confluent**: Yes, trivially confluent.
- **Unique Normal Forms**: Yes, trivially has unique normal forms because there are no elements or rewrites.

2. $A = \{a\}$ **and** $R = \emptyset$

- **Terminating**: Yes, trivially terminating because there are no relations to rewrite a.
- **Confluent**: Yes, trivially confluent.
- **Unique Normal Forms**: Yes, a is its own normal form since there are no rewrites.

3. $A = \{a\}$ **and** $R = \{(a, a)\}$

- **Terminating**: No, $a \to a$ is a loop, so there is an infinite rewrite sequence.
- **Confluent**: Yes, trivially confluent because there is only one element with a single rewriting path.
- **Unique Normal Forms**: No, there is no unique normal form since a can be rewritten infinitely to itself.

4. $A = \{a, b, c\}$ **and** $R = \{(a, b), (a, c)\}$

- **Terminating**: Yes, terminating because each element can be rewritten only a finite number of times.
- **Confluent**: No, not confluent because a can be rewritten to either b or c, leading to different results.
- **Unique Normal Forms**: No, there is no unique normal form as a can reach either b or c.

5. $A = \{a, b\}$ **and** $R = \{(a, a), (a, b)\}$

- **Terminating**: No, because $a \to a$ creates an infinite loop.
- **Confluent**: Yes, trivially confluent as there are only two elements and no branching rewrites.

- **Unique Normal Forms**: No, a does not have a unique normal form due to the loop.

6. $A = \{a, b, c\}$ **and** $R = \{(a, b), (b, b), (a, c)\}$

- **Terminating**: No, because $b \to b$ is a loop, allowing infinite rewrites.
- **Confluent**: No, not confluent because a can rewrite to either b or c, leading to different results.
- **Unique Normal Forms**: No, since a can reach different results (either b or c).

7. $A = \{a, b, c\}$ **and** $R = \{(a, b), (b, b), (a, c), (c, c)\}$

- **Terminating**: No, both $b \to b$ and $c \to c$ create loops.
- **Confluent**: No, not confluent because a can rewrite to either b or c, leading to different results.
- **Unique Normal Forms**: No, since a can reach different results (either b or c).

Images for the above ARSs

1. Confluent: True. Terminating: True. Has unique normal forms: True.
2. Confluent: True. Terminating: True. Has unique normal forms: False.
3. Confluent: True. Terminating: False. Has unique normal forms: True.
4. Confluent: True. Terminating: False. Has unique normal forms: False.
5. Confluent: False. Terminating: True. Has unique normal forms: True.
6. Confluent: False. Terminating: True. Has unique normal forms: False.
7. Confluent: False. Terminating: False. Has unique normal forms: True.
8. Confluent: False. Terminating: False. Has unique normal forms: False.

1. $A \xrightarrow{*} B$

2. $A\downarrow \xrightarrow{*} B \qquad A \xrightarrow{*} C$

3. $A \longrightarrow B \qquad A \longrightarrow C \qquad B \xrightarrow{*} D \qquad C \xrightarrow{*} D$

4. $A \xrightarrow{*} B \qquad A \xrightarrow{*} C$

5. $A \longrightarrow A\downarrow$

6. $A \longrightarrow A \qquad A \xrightarrow{*} B$

7. $A \longrightarrow A\downarrow \qquad A \xrightarrow{*} B$

8. $A \xrightarrow{*} B \qquad B \longrightarrow B \qquad A \xrightarrow{*} C \qquad C \longrightarrow C$

Comments and Questions

After working with ARSs for this week, are we able to implement any recursive functions that have the ability to solve all of problems that we have encountered so far or are there too many exceptions for that to be the case?

3 Lessons from the Assignments

(Delete and Replace): Write three pages about your individual contributions to the project.

On 3 pages you describe lessons you learned from the project. Be as technical and detailed as possible. Particularly valuable are *interesting* examples where you connect concrete technical details with *interesting* general observations or where the theory discussed in the lectures helped with the design or implementation of the project.

Write this section during the semester. This is approximately a quarter of apage per week and the material should come from the work you do anyway. Just keep your eyes open for interesting lessons.

Make sure that you use LaTeX to structure your writing (eg by using subsections).

4 Conclusion

(Delete and Replace): (approx 400 words) A critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of software engineering? What did you find most interesting or useful? What improvements would you suggest?

References

[BLA] Author, Title, Publisher, Year.