CPSC-354 Report

Yoni Kazovsky
Chapman University

December 15, 2024

Abstract

This abstract is merely temporary, and will be updated upon completion of the report. This report will give an overview on the basic foundations of CPSC 354, Programming Languages. This includes notes, homework assignments, and exploration of course topics.

Contents

1 Introduction

...

2 Week by Week

2.1 Week 1

Notes

In Week 1 of CPSC 354, we explored the concept of mathematical proofs, defined as logical reasoning steps that interact with the user rather than merely stating correctness. We began with a reading from *Gödel,

Escher, Bach: an Eternal Golden Braid* by Douglas Hofstadter, focusing on the "MU-puzzle," a formal system challenge involving rules for manipulating the letters M, I, and U to produce "MU" from "MI." The puzzle introduces theorems, derivation, and the distinction between being "inside" (following system rules) and "outside" (analyzing the system externally). While humans and machines differ in their approach—machines mechanically repeat rules, humans discern patterns and gain insight, highlighting human cognition's ability to transcend formal systems.

Homework

NNG Tutorial World Level 5:

a+(b+0)+(c+0)=a+b+c.

Solution:

First we use the Lean add zero proof to remove the 0 in b+0.

Then we use the Lean add zero proof to remove the 0 in c+0.

Finally we are left with a+b+c = a+b+c and we can use rfl to confirm our proof with reflexive property.

...

NNG Tutorial World Level 6:

a+(b+0)+(c+0)=a+b+c.

Solution:

First we use the Lean precision add zero proof (tarteting c) to remove the 0 in c+0.

Then we use the Lean add zero proof to remove the remaining 0 in b+0.

Finally we are left with a+b+c = a+b+c and we can use rfl to confirm our proof with reflexive property.

...

NNG Tutorial World Level 7:

For all natural numbers a, we have succ(a)=a+1.

Solution:

First we unwravel the one with the Lean rw proof to eliminate the one with a succ0.

Then we use the Lean rw proof with add succ to change n + succ0 into succ(n+0).

Then we use the rw proof to rewrite succ(n+0) into succ(n).

Finally we are left with succ(n) = succ(n) and we can use rfl to confirm our proof with reflexive property.

...

NNG Tutorial World Level 8:

For all natural numbers a, we have succ(a)=a+1.

Solution:

For this problem I simplified both sides of the equation using the rewrite proof with succesors such as 3 = succ 2 and more.

Eventually I got to this point: succ (succ 0) + succ (succ 0) = succ (succ (succ (succ 0)))

At this point I began using rw add succ to simplify the left side of the equation

Then I used rw add zero to remove a remaining zero and I was left with this: succ (succ (succ (succ 0))) = succ (succ (succ (succ 0)))

At this point I used the rfl to confirm my proof reflexively.

...

In all of the above examples I used the Lean rfl proof which directly corresponds to the mathematical reflexive property which states that: any number a is equal to itself. In other words a = a or b+c = b+c, etc.

Comments and Questions

This section was a good refresh on some of the discrete mathematics concepts that I had forgotten over break. My question for this week is the following:

How are these mathematical concepts applied to the developement of programming languages? How can the lessons we learn from the mu puzzle be applied to real-world problems pertaining to the developement of computer programs and languages?

2.2 Week 2

Notes

In the readings and class lectures about recursion and the towers of hanoi, I learned about what recursion is and how it can be used to optimize certain sysetems. This is evident in the towers of hanoi where we use a recursive formula or pattern to complete the challenge in the minimum amount of steps.

This was a good refresh on discrete mathematics and I look forward to learning how to apply the use of recursion in this course and in other challenges that can be solved mathematically.

We also addressed student questions from the Discord server, discussing recursion's role in simplifying problems, the challenges of AI correcting code typos due to ambiguity, and AI's influence on programming language development through code synthesis and natural language translation.

Homework

NNG Addition World Level 1:

For all natural numbers n, we have 0 + n = n.

Solution:

1. induction n with d hd
2. rw add zero
3. rfl
4. rw add succ
5. rw hd
6. rfl

...

NNG Addition World Level 2:

For all natural numbers a,b, we have succ(a)+b=succ(a+b)

Solution:

1. induction b
2. rw add zero
3. rfl
4. rw add succ
5. rw add succ
6. rw n ih
7. rfl

...

NNG Addition World Level 3:

On the set of natural numbers, addition is commutative. In other words, if a and b are arbitrary natural numbers, then a+b=b+a

Solution:

1. induction b
2. rw add zero
3. rw zero add
4. rfl
5. rw add succ
6. rw succ add
7. rw n ih
8. rfl

...

NNG Addition World Level 4:

On the set of natural numbers, addition is associative. In other words, if a,b and c are arbitrary natural numbers, we have (a+b)+c=a+(b+c).

Solution:

1. induction a
2. rw zero add
3. rw zero add
4. rfl
5. rw succ add
6. rw succ add
7. rw succ add
8. rw n ih
9. rfl

...

NNG Addition World Level 5:

If a,b and c are arbitrary natural numbers, we have (a+b)+c=(a+c)+b.

Solution:

1. induction a
2. rw zero add
3. rw zero add
4. rw add comm
5. rfl
6. rw succ add
7. rw succ add
8. rw succ add
9. rw succ add
10. rw n ih
11. rfl

This lean proof is similar to the corresponding mathematical proof in the way that it uses induction to prove the theory on a single smaller equation and applies the inductive step and uses said proof to prove a property across all equations!

Comments and Questions

This section was yet another good refresh on discrete and reintroduced me to the concept of induction in proofs. I also loved the towers of hanoi activity and its modeling of recursive processes! My question is how similar puzzles can appear in programming languages and specifically the creation of them? How can we apply lean proofs to such challenges and how will this look in the form of actual code?

2.3 Week 3

Notes

This week we began developing our calculator in python which has proven to be more difficult than expected. I believe that the idea is to use a recursive method to scan an input for the subsections of a more complicated long input. On Tuesday, we explored recursion through a question about developing solutions and avoiding misleading patterns. The professor explained the "divide and conquer" strategy for optimizing code by breaking it into smaller sub-problems and introduced tail-call optimization (TCO). TCO minimizes stack space by reusing the current stack frame, making recursion more efficient. This week's reading, "The Location of Meaning," illustrated the distinction between information-bearers, like vinyl grooves, and information-revealers, like a record player. The metaphor highlights how meaning is context-dependent, requiring the right mechanism to transform dormant information into something understandable.

Homework

Homework on Github

Comments and Questions

How can Large Language Models (LLMs), like ChatGPT, be effectively integrated into a programming languages course to enhance learning while avoiding over-reliance, particularly for abstract concepts like recursion, lambda calculus, and type systems?

2.4 Week 4

Notes

This week we continued to work on the python project which proved to become much more difficult now that I was trying to implement methods that allowed me to breakdown an expression and parse it into multiple subsections with operators and operands. In addition we learned about parsing and parsing trees which proved to be a very useful concept in implementing my python calculator.

Homework

1. 2 + 1

Exp→Exp+Exp1

Exp→Exp1

Exp1→Exp2

Exp2→Integer

Exp1→Exp2

Exp2→Integer

—

—

2. 1 + 2 * 3

Exp→Exp+Exp1

Exp→Exp1

Exp1→Exp2

Exp2→Integer

Exp1→Exp1*Exp2

Exp1→Exp2

Exp2→Integer

Exp2→Integer

—

3. 1 + (2 * 3)

Exp→Exp+Exp1

Exp→Exp1

Exp1→Exp2

Exp2→Integer

Exp1→Exp2

Exp2→(Exp)

Exp→Exp1

Exp1→Exp1*Exp2

Exp1→Exp2

Exp2→Integer

Exp2→Integer

—

4. (1 + 2) * 3

Exp→Exp1

Exp1→Exp1*Exp2

Exp1→Exp2

Exp2→(Exp)

Exp→Exp+Exp1

Exp→Exp1

Exp1→Exp2

Exp2→Integer

Exp1→Exp2

Exp2→Integer

Exp2→Integer

—

5. 1 + 2 * 3 + 4 * 5 + 6

Exp→Exp+Exp1

Exp→Exp+Exp1

Exp→Exp1

Exp1→Exp2

Exp2→Integer

Exp1→Exp1*Exp2

Exp1→Exp2

Exp2→Integer

Exp2→Integer

Exp1→Exp1*Exp2

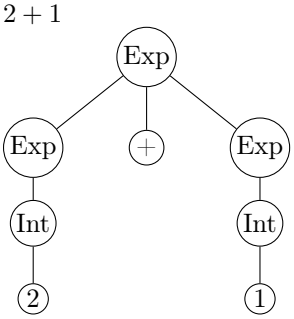
Exp2→Integer

Exp2→Integer

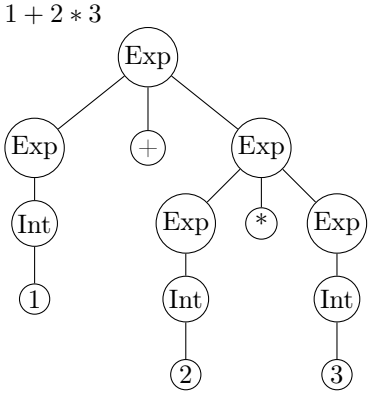
Exp2→Integer

TREES:

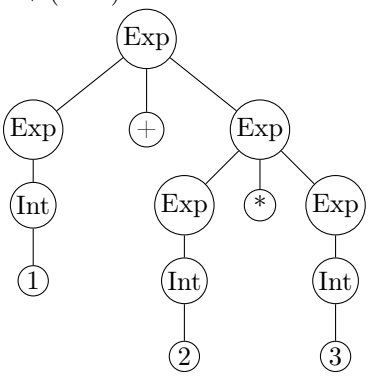
$2 + 1$



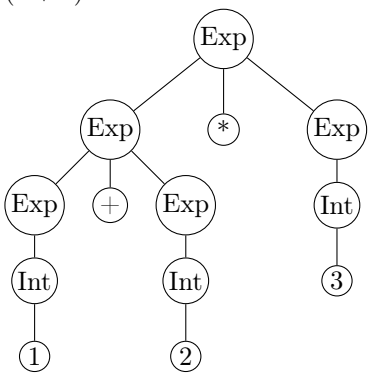
$1 + 2 * 3$



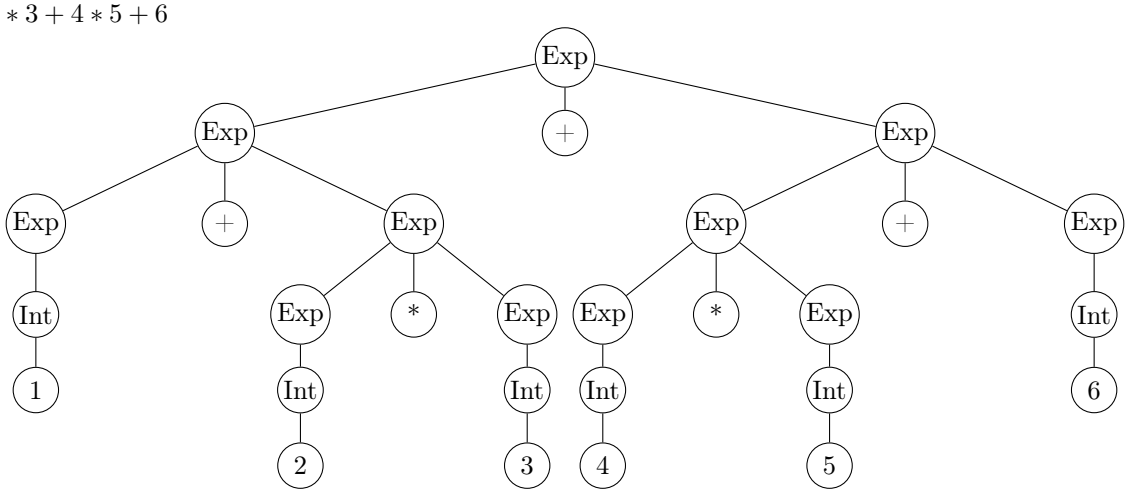
$1 + (2 * 3)$



$(1 + 2) * 3$



$1 + 2 * 3 + 4 * 5 + 6$



Comments and Questions

My question for this week is the following: How can this concept of parcing be applied to the developement of a programming language, specifically in the ways in which the backend handles order of operations regarding actual code as opposed to just calculations like we did in our calculators?

2.5 Week 5

Notes

This week, we began discussing logical propositions and operations, starting with a real-world example: Proposition P ("It is raining") and Proposition Q ("The street is wet"). Using Lean, we explored how logical reasoning principles can provide proofs for statements like ‘$n = 0 \to n + 1 = 1$‘. We examined propositions as data types that can be true or false, with examples showing how conjunctions ($P \wedge Q$) require evidence, or "witnesses," to prove both components. Through diagrams and rules like formation, introduction, and elimination, we deepened our understanding of logical systems. The readings supported these concepts, introducing Gödel, Escher, Bach's explanation of propositional calculus and Lean's role as both a functional programming language and a proof assistant. They also explored propositional and predicate logic, covering syntax, connectives, inference rules, and the validation of logical arguments, emphasizing formal logic's foundational role in programming languages and mathematics.

Homework

Level 1

```
variables P : Prop

-- Assume that P is on a to-do list, and we need to prove P.
example (todo_list : P) : P :=
begin
  -- Directly use the given assumption to complete the proof.
  exact todo_list,
end
```

Level 2

```
variables P S : Prop
example (p : P) (s : S) : P ∧ S :=
begin
   -- Use the and.intro rule to construct the conjunction.
   exact and.intro p s,
 end
```

Level 3

```
variables A I O U : Prop
example (a : A) (i : I) (o : O) (u : U) : (A ∧ I) ∧ O ∧ U :=
begin
  -- Combine the propositions step by step using nested and.intro.
  exact ⟨⟨a, i⟩, ⟨o, u⟩⟩,
end
```

Level 4

```
variables P S : Prop
example (vm : P ∧ S) : P :=
begin
  -- Use and.left to extract P from the conjunction.
  have p := and.left vm,
  -- Conclude the proof with the extracted value.
  exact p,
end
```

Level 5

```
variables P Q : Prop
example (h : P ∧ Q) : Q :=
begin
  -- Use and.right to extract Q from the conjunction.
  exact h.right,
end
```

Level 6

```
variables A I O U : Prop
example (h1 : A ∧ I) (h2 : O ∧ U) : A ∧ U :=
begin
  -- Extract A from h1.
  have a := h1.left,
  -- Extract U from h2.
  have u := h2.right,
  -- Combine A and U using and.intro.
  exact and.intro a u,
end
```

Level 7

```
variables C L : Prop
example (h : (L ∧ ((L ∧ C) ∧ L) ∧ L ∧ L ∧ L) ∧ (L ∧ L) ∧ L) : C :=
begin
  -- Extract the left-hand side of the conjunction.
  have h₁ := h.left,
  -- Navigate through the structure to isolate C.
  exact h.left.right.left.left.right,
end
```

Level 8

```
variables A C I O P S U : Prop
example (h : ((P ∧ S) ∧ A) ∧ ¬I ∧ (C ∧ ¬O) ∧ ¬U) : A ∧ C ∧ P ∧ S :=
begin
  -- Extract the conjunction (P ∧ S) ∧ A.
  have h₁ := h.left,
  -- Separate P ∧ S and A.
  have ps := h₁.left,
  have a := h₁.right,
  -- Extract (C ∧ ¬O) and isolate C.
  have h₂ := h.right.right.left,
```

```
  have c := h₂.left,
  -- Combine A, C, P, and S using and.intro.
  exact ⟨a, c, ps.left, ps.right⟩,
end
```

Mathematical Proof for Level 8:

$$
\begin{aligned}
&= ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U && \text{assumption} \\
&= (P \wedge S) \wedge A && \text{and\_left (1)} \\
&= P \wedge S && \text{and\_left (2)} \\
&= A && \text{and\_right (2)} \\
&= C \wedge \neg O && \text{and\_right (1)} \\
&= C && \text{and\_left (5)} \\
&= P && \text{and\_left (3)} \\
&= S && \text{and\_right (3)} \\
&= A \wedge C \wedge P \wedge S && \text{and\_intro (4) (6) (7) (8)}
\end{aligned}
$$

Comments and Questions

How can Lean logic proofs enhance our understanding of programming languages, particularly in verifying program correctness and exploring foundational concepts like recursion, type systems, and propositions?

2.6 Week 6

Notes

This week, we explored lambda calculus, its syntax, and its foundational role in programming languages. Lambda calculus simplifies functions into three constructs: lambda abstraction (e.g., $\lambda x.x + 1$), evaluation (e.g., $(\lambda x.x + x)2 \rightsquigarrow 4$), and variables, focusing on computation without named functions. We discussed abstraction principles, alpha equivalence, and De Bruijn indices, highlighting the importance of variable substitution and avoiding conflicts via capture-avoiding substitution. Computation in lambda calculus was also compared to programming, emphasizing its non-symmetric nature and the precedence rules for left-associative applications.

Homework

Levels 1-8 of the "→ Tutorial: Party Snacks" from the Lean Logic Game.

Level 1

```
-- Show that the cake will be delivered to the party.
variables P C : Prop
example (p : P) (bakery_service : P → C) : C :=
begin
  -- Use the bakery service to derive C (the cake) from P.
  exact (bakery_service p),
end
```

Level 2

```
-- Prove that a proposition implies itself (identity property).
variables C : Prop
```

```
example : C → C :=
begin
  -- Use the identity function to establish C → C.
  exact λ h : C ↦ h
end
```

Level 3

```
-- Demonstrate that conjunction (∧) is commutative.
variables I S : Prop
example : I ∧ S → S ∧ I :=
begin
  -- Use and_intro to swap h.right (S) and h.left (I).
  exact λ h => and_intro (h.right) (h.left),
end
```

Level 4

```
-- Prove the transitivity of implication (→).
variables C A S : Prop
example (h1 : C → A) (h2 : A → S) : C → S :=
begin
  -- Apply h1 to derive A, then use h2 to obtain S.
  exact λ h => h2 (h1 h)
end
```

Level 5

```
-- Prove the chain of reasoning to derive U.
variables P Q R S T U : Prop
example (p : P) (h1 : P → Q) (h2 : Q → R) (h3 : Q → T) (h4 : S → T) (h5 : T → U) : U :=
begin
  -- Use h1 to derive Q from P, apply h3 to get T, and h5 to get U.
  exact h5 (h3 (h1 p)),
end
```

Level 6

```
-- Explore how conjunction interacts with implication.
variables C D S : Prop
example (h : C ∧ D → S) : C → D → S :=
begin
  -- Assume C and D, then use h on the pair ⟨c, d⟩.
  exact λ c d => h ⟨c, d⟩
end
```

Level 7

```
-- Reverse the interaction between conjunction and implication.
variables C D S : Prop
example (h : C → D → S) : C ∧ D → S :=
begin
  -- Assume ⟨c, d⟩ is C ∧ D, then use h with c and d.
  exact λ ⟨c, d⟩ => h c d
end
```

Level 8

```
-- Show that implication distributes over conjunction.
variables C D S : Prop
example (h : (S → C) ∧ (S → D)) : S → C ∧ D :=
begin
  -- Assume S, then use h.left for C and h.right for D.
  exact λ s => ⟨h.left s, h.right s⟩,
end
```

Comments and Questions

My question for this week is how can some of the concepts we learned with different types of proofs be applied to lamda calculus proofs. And how can these lamda calculus proofs then be used in the developement of a programming language?

2.7 Week 7

Notes

This week, we explored key concepts in lambda calculus, focusing on currying, uncurrying, logic encoding, and Church numerals. Starting with the goal of transforming $h : (A \wedge B) \to C$ into $h' : A \to (B \to C)$, we demonstrated how to abstract over variables using lambda notation, such as $h' := \lambda a.\lambda b.h(a, b)$. Similarly, we reversed the process by defining $k' := \lambda p.k(p.left, p.right)$ for uncurrying. Examples like addition for natural numbers illustrated the correspondence between curried and uncurried forms.

Next, we introduced constructs for logic encoding, such as true, false, and if-then-else. These were defined as true $:= \lambda x.\lambda y.x$, false $:= \lambda x.\lambda y.y$, and if-then-else $:= \lambda x.\lambda y.\lambda z.xyz$. Logical operations like not, and, or were expressed using these constructs. For instance, not $:= \lambda x.x$ false true, and $:= \lambda x.\lambda y.x\, y$ false, and or $:= \lambda x.\lambda y.x$ true y.

We explored evaluation strategies: applicative order (evaluating inner expressions first) and normal order (reducing outermost expressions first). Normal order ensures termination if possible but may be slower. Finally, Church numerals were introduced to encode arithmetic, with definitions like $1 := \lambda f.\lambda x.fx$, $2 := \lambda f.\lambda x.f(fx)$, and $0 := \lambda f.\lambda x.x$. Arithmetic operations, such as addition, were built upon this framework.

The week concluded with exercises evaluating expressions like and true false, using substitution and beta reduction to demonstrate the equivalence of these constructs and reinforce their logical consistency.

Homework

1. Simplify the following lambda expression:

```
((λm.λn. m n) (λf.λx. f (f x))) (λf.λx. f (f (f x)))
```

The reduction proceeds in 7 steps using beta reduction:

```
((λm.λn. m n) (λf.λx. f (f x))) (λf.λx. f (f (f x)))
⇝ (λn. (λf.λx. f (f x)) n) (λf.λx. f (f (f x)))     -- Replace m with (λf.λx. f (f x))
⇝ (λn. (λf.λx. f (f x)) (λf.λx. f (f (f x))))       -- Replace n with (λf.λx. f (f (f x)))
⇝ (λf.λx. f (f (f x)))                             -- Apply (λf.λx. f (f x))
⇝ (λf.λx. f (f (f (f (f x)))))                     -- Result: Church numeral 9
```

2. Describe the natural number function implemented by $(\lambda m.\lambda n.mn)$**:**

This lambda function represents application. It takes two arguments: a "function" `m` and an "element" `n`, then applies `m` to `n` exactly once.

Comments and Questions

How does the process of simplifying lambda terms using beta reduction help us understand the computational behavior of functional programming languages?

2.8 Week 8 and 9

Notes

In these weeks, we explored advanced extensions of lambda calculus, building on earlier topics like boolean logic, conditional constructs, and arithmetic operations such as numerals, addition, and exponentiation. A key focus was the introduction of the 'let' construct, which enables the use of local variable names and supports recursion. Recursion, the ability of a function to reference itself, relies on 'let' for implementation in the absence of explicit function names. This highlights how 'let' expressions facilitate recursive behavior in lambda calculus. We also worked on developing a lambda calculus interpreter. This taught me a lot about lambda calculus and how it is implemented in programming languages.

Homework

Excercises 2-8:

2. Explain why a b c d reduces to (((a b) c) d) and why (a) reduces to a:

In lambda calc. expressions such as a b c d are evaluated as nested applications because the function application is inheretly left-associative. In tha case of (a), parenthesis in lambda calculus are used for grouping 2 or more variables so since a is just one variable the parenthesis can be dropped leaving us with just a.

3. How does capture avoiding substitution work? Investigate both by making relevant test cases and by looking at the source code. How is it implemented?

Capture-avoiding substitution prevents accidental overlaps of variable names when replacing variables in an expression by renaming certain variables to keep them distinct. This process ensures that each variable keeps its intended meaning, even if it needs a new name to avoid conflicts during substitution. It is implemented by checking each variable before the substitution occurs

4. Do you always get the expected result? Do all computations reduce to normal form?

I got the expected result sometimes. Not all computations reduce to normal form because some lambda expressions can cause infinite loops.

5. What is the smallest lambda expression you can find (minimal working example, MWE) that does not reduce to normal form?

The smallest expresion that does not reduce to normal form is (lambdax.x x) (lambdax.x x)

6. no question asked here

7. How does the interpreter evaluate ((lambdam.lambdan. m n) (lambdaf.lambdax. f (f x))) (lambdaf.lambdax. f (f (f x)))? Do a calculation similarly to when you evaluated ((lambdam.lambdan. m n) (lambdaf.lambdax. f (f x))) (lambdaf.lambdax. f (f (f x))) for the homework, but now follow precisely the steps taken by interpreter.py

Using the input:"((lambdam.lambdan. m n) (lambdaf.lambdax. f (f x))) (lambdaf.lambdax. f (f (f x)))"

Substitute called: replacing m with (lambdaf.(lambdax.(f (f x)))) in (lambdan.(m n))

Proceeding with substitution in lambda body for n

Substitute called: replacing m with (lambdaf.(lambdax.(f (f x)))) in (m n)

Substitute called: replacing m with (lambdaf.(lambdax.(f (f x)))) in m

Substituting variable m with (lambdaf.(lambdax.(f (f x))))

Substitute called: replacing m with (lambdaf.(lambdax.(f (f x)))) in n

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in ((lambdaf.(lambdax.(f (f x)))) n)

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in (lambdaf.(lambdax.(f (f x))))

Proceeding with substitution in lambda body for f

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in (lambdax.(f (f x)))

Proceeding with substitution in lambda body for x

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in (f (f x))

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in f

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in (f x)

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in f

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in x

Substitute called: replacing n with (lambdaf.(lambdax.(f (f (f x))))) in n

Substituting variable n with (lambdaf.(lambdax.(f (f (f x)))))

Substitute called: replacing f with (lambdaf.(lambdax.(f (f (f x))))) in (lambdax.(f (f x)))

Proceeding with substitution in lambda body for x

Substitute called: replacing f with (lambdaf.(lambdax.(f (f (f x))))) in (f (f x))

Substitute called: replacing f with (lambdaf.(lambdax.(f (f (f x))))) in f

Substituting variable f with (lambdaf.(lambdax.(f (f (f x)))))

Substitute called: replacing f with (lambdaf.(lambdax.(f (f (f x))))) in (f x)

Substitute called: replacing f with (lambdaf.(lambdax.(f (f (f x))))) in f

Substituting variable f with (lambdaf.(lambdax.(f (f (f x)))))

Substitute called: replacing f with (lambdaf.(lambdax.(f (f (f x))))) in x (lambdax.((lambdaf.(lambdax.(f (f (f x))))) ((lambdaf.(lambdax.(f (f (f x))))) x)))

8. Write out the trace of the interpreter in the format we used to picture the recursive trace of hanoi. Only write lines that contain calls to evaluate() or calls tosubstitute()[4]. Add the line numbers

12: eval(('app', ('app', ('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n')))), ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x')))))), ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x'))))))))

12: eval(('app', ('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n')))), ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x')))))))

12: eval(('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n')))))

51: Returning ('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n'))))

39: substitute(('lam', 'n', ('app', ('var', 'm'), ('var', 'n'))), m, ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x'))))))

60: substitute(('lam', 'n', ('app', ('var', 'm'), ('var', 'n'))), m, ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x'))))))

60: substitute(('app', ('var', 'm'), ('var', 'n')), m, ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x'))))))

60: substitute(('var', 'm'), m, ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x'))))))

64: Returning ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x')))))

60: substitute(('var', 'n'), m, ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x'))))))

64: Returning ('var', 'n')

Comments and Questions

Week 8: How can we use lambda calculus to perform tasks in different orders when managing a tasks in a programming language?

Week 9: What implications does lambda calculus have on the functionality of programming languages? How is it implemented? Is there an easier and more efficient implementing capture-avoiding substitution in the interpreter?

2.9 Week 10

Notes

This week, we explored rewriting theory, which transforms expressions into new forms and underpins computation in various fields. We discussed rewrite rules in systems like Turing machines and their application to processes such as the "MU game" and the Collatz sequence. Rewriting systems are formalized as (A, R), where A is a set of terms and R defines transformations, often visualized as directed graphs. Key properties include confluence, ensuring all reduction paths lead to the same result, and termination, guaranteeing computations eventually stop. Normal forms, representing irreducible terms, play a crucial role in this framework. Practical examples, such as bubble sort, highlight rewriting's ability to formalize algorithms and ensure predictable outcomes in computation.

Homework

Questions of Reflection from Assignment 3

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

The most challenging part was implementing capture-avoiding substitution in the interpreter, especially ensuring that variable renaming didn't accidentally change the meaning of expressions. Tracking variable scopes and managing recursive evaluations required careful debugging and a solid understanding of lambda calculus mechanics, which was conceptually and technically demanding.

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

The key insight came from breaking down each component of the interpreter into smaller, testable parts. By isolating the substitution mechanism and testing it with minimal expressions, I could see how substitutions worked in different cases. This modular approach helped me catch issues early on and better understand the flow of lambda expressions through the interpreter.

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

The most interesting takeaway was realizing how lambda calculus can be used to simulate computation purely through function applications, without any inherent notion of numbers, operators, or other primitives.

Building an interpreter from scratch gave me a deeper appreciation of how abstract computation principles can be directly implemented in code, connecting theory with practice in a very hands-on way

Comments and Questions

How do the properties of confluence and termination in rewriting systems impact the reliability and predictability of algorithms, such as sorting algorithms or computational models like Turing machines? What is the application of interpreters like this in real-world systems?

2.10 Week 11

Notes

This week, we discussed confluence and termination in abstract rewriting systems (ARS), defined as a pair (A, R) where A is a set and R a binary relation. Confluence ensures that computations starting from the same element can always reach a common result, which is critical for guaranteeing consistency. Termination, or strict normalization, ensures no infinite computation paths exist, whereas weak normalization only guarantees that every element has a normal form, even if some paths are infinite. For example, a rewrite system with rules like $aa \to a$ or $ab \to b$ may always terminate, reducing to a single normal form. However, systems like $a \to a$ and $a \to b$ highlight that weak normalization does not imply termination, as infinite loops can occur. Conversely, terminating systems are always normalizing since all computations end in normal forms. These concepts underpin programming languages, aiding in understanding evaluation consistency and efficiency.

Homework

1. $A = \emptyset$

This ARS represents an empty set

1. Yes ARS terminating.
2. Yes ARS confluent.
3. Yes has unique normal forms.

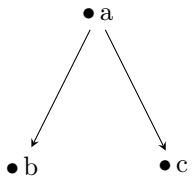
2. $A = \{a\}, R = \emptyset$

1. Yes ARS terminating.
2. Yes ARS confluent.
3. Yes has unique normal forms.

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

1. No ARS not terminating.
2. Yes ARS confluent.
3. No does not have unique normal forms.

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

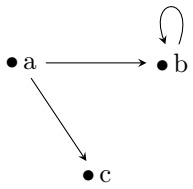


1. Yes ARS terminating.
2. No ARS not confluent.
3. No does not have unique normal forms.

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

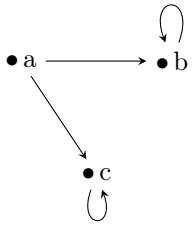
1. No ARS not terminating.
2. Yes ARS confluent.
3. No does not have unique normal forms.

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



1. No ARS not terminating.
2. Yes ARS confluent.
3. No does not have unique normal forms.

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



1. No ARS not terminating.
2. No ARS not confluent.
3. No does not have unique normal forms. “‘

Comments and Questions

How can ARS's improve the reliability and efficiency of modern programming languages in verifying complex code bases, especially in safety-critical applications like autonomous vehicles or financial systems?

2.11 Week 12

Notes

This week, we reviewed key concepts of Abstract Rewriting Systems (ARS). An ARS $(A, \to)$ consists of a set A (e.g., terms or expressions) and a binary relation $\to$ that defines how elements can be rewritten. Key properties include termination (no infinite rewrite sequences), confluence (any two rewrite paths from the same starting point can join at a common successor), normalization (every element has at least one normal form), and unique normal forms (if normal forms exist, they are unique). Confluence does not guarantee normalization, but it ensures that any existing normal forms are unique. Examples of confluence include bubble sort, where the rule $ba \to ab$ ensures a sorted, unique result (e.g., ab), lambda calculus, where beta reduction produces unique outcomes, and calculator arithmetic, which yields consistent results when following precedence rules. The Church-Rosser property extends confluence by ensuring that equivalent elements can join at a common successor. Confluence is critical in programming for deterministic outputs, and invariants, such as maintaining a consistent result during computation, help verify properties like termination and confluence.

Homework

Exercise 1

Rewrite rule: $ba \to ab$

1. **Why does the ARS terminate?** The ARS terminates because each rewrite step decreases the number of inversions in the string, eventually reaching a sorted form.
2. **What is the result of a computation (the normal form)?** The normal form is the string where all a's precede all b's.
3. **Show that the result is unique (the ARS is confluent).** Confluence holds because any two paths lead to the same sorted result.
4. **What specification does this algorithm implement?** The ARS implements a sorting algorithm that orders a's before b's.

Exercise 2

Rewrite rules:

$$aa \to a,$$
$$bb \to a,$$
$$ab \to b,$$
$$ba \to b$$

1. **Why does the ARS terminate?** The ARS terminates because each rewrite step reduces the length of the string.
2. **What are the normal forms?** The normal forms are either a or b.
3. **Is there a string s that reduces to both a and b?** No, the ARS is confluent, so each string reduces to a unique normal form.
4. **Replacing $\to$ with $=$, which words become equal?** Any string with an even number of a's and b's reduces to a, and strings with an odd number reduce to b.
5. **Which specification does the algorithm implement?** The ARS determines the parity of the total count of a's and b's, with a representing even and b representing odd.

Exercise 3

Rewrite rules:

$$aa \to a,$$
$$bb \to b,$$
$$ab \to ba,$$
$$ba \to ab$$

1. **Why does the ARS not terminate?** The rules $ab \to ba$ and $ba \to ab$ create an infinite loop for certain strings.
2. **What are the normal forms?** There are no normal forms due to the non-termination.
3. **Modify the ARS to make it terminating with unique normal forms:** Add $ab \to a$ or $ba \to b$ to break the loop and ensure termination.
4. **What specification does the ARS implement?** It simplifies strings by collapsing repeated letters and ordering a's and b's deterministically.

Exercise 4

Rewrite rules:

$$ab \to ba,$$
$$ba \to ab$$

1. Same questions as Exercise 1. This ARS represents a basic sorting algorithm where the strings are reordered to have all a's followed by all b's.

Exercise 5

Rewrite rules:

$$ab \to ba,$$
$$ba \to ab,$$
$$aa \to \varepsilon,$$
$$b \to \varepsilon$$

1. **Reduce some example strings such as $abba$ and $bababa$.**
   - $abba \to baba \to abab \to aabb \to \varepsilon$
   - $bababa \to ababab \to aabbab \to aababb \to \varepsilon$
2. **Why is the ARS not terminating?** The rules $ab \to ba$ and $ba \to ab$ create infinite loops.
3. **How many equivalence classes does $\leftrightarrow^*$ have? Can you describe them?** Two equivalence classes: strings reducing to ε and those that do not.
4. **Can you change the rules to make the ARS terminating without changing equivalence classes?** Modify $ab \to \varepsilon$ or $ba \to \varepsilon$ to eliminate infinite loops.
5. **Write questions about strings:** What is the shortest representation of a string? Does every string reduce to ε?

Exercise 5b

Rewrite rules:

$$ab \to ba,$$
$$ba \to ab,$$
$$aa \to a,$$
$$b \to \varepsilon$$

1. **Reduce some example strings such as $abba$ and $bababa$.**
   - $abba \to baba \to abab \to aabb \to ab \to ba \to a$
   - $bababa \to ababab \to aabbab \to aabab \to abab \to aabb \to ab \to a$

   Under these rules, strings reduce to either a or ε, depending on the counts of a's and b's.
2. **Why is the ARS not terminating?** The rules $ab \to ba$ and $ba \to ab$ can cause infinite loops if not further constrained.
3. **How many equivalence classes does $\leftrightarrow^*$ have? Can you describe them?**
   - There are two equivalence classes: strings that reduce to a and those that reduce to ε.
   - These equivalence classes are determined by the parity of b's in the original string: even b's reduce to a, and odd b's reduce to ε.
4. **Can you change the rules to make the ARS terminating without changing equivalence classes?**
   - Modify $ab \to \varepsilon$ or $ba \to \varepsilon$ to eliminate potential infinite loops caused by cyclic rewrites.
   - For instance, adding a rule $ab \to a$ would force the system to terminate while preserving the equivalence classes.
5. **Write questions about strings:**
   - How many steps are required for a string to reach its normal form?
   - Given a string, does it always reduce to a or ε, and how can we predict the result?

Comments and Questions

How does the concept of confluence in Abstract Rewriting Systems (ARS) ensure consistency in computational processes, and why is it critical for programming languages to maintain deterministic outputs? Provide examples, such as bubble sort or lambda calculus, to illustrate your points.

2.12 Week 13

Notes

This week, we explored invariants through a chessboard activity, showing the original 8x8 board could be covered using 32 tiles, but the mutilated board (missing two blue tiles) could not, as it breaks the invariant requiring equal white and blue tiles in each step. The normal forms were either a fully covered board or two isolated uncovered tiles, demonstrating a terminating ARS. We then discussed defining functions using `let` and recursion. Simple definitions like `let succ =` λ`n.n+1` work directly, but recursive definitions like factorials require `let rec`, which ensures proper self-referencing through fixed-point combinators. Key reductions include `(let x = e1 in e2) =` $(\lambda$`x.e2)e1` and `(let rec f = e1 in e2) = (fix(`λ`f.e2))e1`.

Homework

Initial expression:

$$\text{let rec fact} = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \text{fact } (n-1) \text{ in fact } 3$$

1. **Step-by-step computation:**
   - **Definition of `let rec`:**

     $$\text{let rec fact} = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \text{fact } (n-1) \text{ in fact } 3$$

     becomes

     $$\text{let fact} = (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1))) \text{ in fact } 3.$$

   - **Definition of `let`:**

     $$\text{let fact} = (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1))) \text{ in fact } 3$$

     becomes

     $$\big((\lambda \text{fact. fact } 3)\ (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))\big).$$

   - **β-reduction (substitute `fix F`):**

     $$\big((\lambda \text{fact. fact } 3)\ (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))\big)$$

     reduces to

     $$\big((\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))\ 3\big).$$

   - **Definition of `fix`:**

     $$\big((\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))\ 3\big)$$

     becomes

     $$\big((\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1))\ (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))\big)(3).$$

   - **Substitute $n = 3$:**

     $$(\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))(n-1))(3)$$

     reduces to

     $$\text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \cdot (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))(2).$$

   - **Definition of `if`:**

     $$\text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \cdot (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))(2)$$

     evaluates to

     $$3 \cdot (\text{fix } (\lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)))(2).$$

   - **Repeat substitution and evaluation for $n = 2, 1, 0$:** The computation proceeds as follows:

     $$3 \cdot (2 \cdot (1 \cdot 1)) = 6.$$

2. **Final result:** The computation of `fact 3` yields:

   $$\boxed{6}$$

Comments and Questions

What are the trade-offs in terms of readability, efficiency, and debugging when using fixed-point combinators compared to traditional recursive function definitions?

3 Lessons from the Assignments

4 Conclusion

References

[BLA] Author, Title, Publisher, Year.