CPSC-354 Report

Your Name
Chapman University

October 13, 2024

Abstract

If will write this abstract... later!

Contents

1 Introduction

This introduction will be filled later, when I know what I want to say!

Grading guidelines (see also below):

- Is typesetting and layout professional?
- Is the technical content, in particular the homework, correct?
- Did the student find interesting references [BLA] and cites them throughout the report?
- Do the notes reflect understanding and critical thinking?
- Does the report contain material related to but going beyond what we do in class?
- Are the questions interesting?

Do not change the template (fontsize, width of margin, spacing of lines, etc) without asking your first.

2 Week by Week

2.1 Week 1

Notes

This week in class we mainly focused on learning Lean, setting up LaTeX, and a brief lecture on the basis of $Proof = Program$. This idea shows how logical mathematical proofs are a constructive process, building on previously founded theorems and definitions. This idea transfers to theoretical programming in the sense that programs are also constructed proofs. The execution of a program is the execution of many logical steps in a proof. I enjoy how this relates to the "human" process as well - our activities are the execution of previously learned strategies in a logical way.

Question

I was interested in learning more about Proof = Program, and some additional sleuthing showed me how a running a program is simply executing the steps in a logical proof. I was wondering how more complex program design methods such as recursion adhere to this idea, and if there are other programming methodologies that do not adhere to Proof = Program?

Homework

Level 5 - Adding Zero

In this level we prove the theorem that **a+0=a** using **a+(b+0)+(c+0)=a+b+c**. Here is how I solved this theorem.

> **repeat rw add zero**
>
> **rfl**

Level 6 - Adding Zero

In this level we built on the solution from the previous level to learn how to use precision rewriting.

> **rw add zero c**
>
> **repeat rw add zero**
>
> **rfl**

Level 7 - Add Suc

In this level we prove the theorem that **succ(a)=a+1**.

> **rw one eq add zero**
>
> **repeat rw add zero**
>
> **rfl**

Level 8 - Add Suc

In this level we prove the equation that **2+2=4**. This was the final level in the Tutorial World, and required the accumulation of definitions and theorems learned so far. I will provide the assumptions that I used when deciding my proof in Lean.

> **rw four eq succ three** —— Any number $n = succ(pred(n))$

rw three eq succ two —— Any number $n = succ(pred(n))$

rw two eq succ one —— Any number $n = succ(pred(n))$

rw one eq succ zero —— Any number $n = succ(pred(n))$

repeat rw add succ —— Using $a + succb = succ(a + b)$

rw add zero —— Using $a + 0 = a$

rfl —— Proves the goal $X = X$

Comments and Questions

Ask at least one **interesting question**[1] on the lecture notes. Also post the question on the Discord channel so that everybody can see and discuss the questions.

2.2 Week 2

Notes

Translating Lean into Math by matching each line in Lean to the corresponding mathematical equation and assumptions. Being able to reverse the proof and translate from Math into Lean is also important. Lean reads from the goal to the axioms, whereas Math is written from the axioms to the goal (usually).

Defining data types recursively (in terms of itself). Syntax varies by language.

Recursion example with the Tower of Hanoi. Breaking logical puzzles into iterative steps, then into recursive steps.

Question

In class we used recursion to prove an algorithmic solution for the tower of hanoi game. Within the scope of proofs, what are the biggest drawbacks/advantages to using iterative vs recursive techniques?

Homework

Level 1 - Zero Add

In this level we prove the theorem that $\mathbf{0 + n = n}$. It was our first use of proof by induction.

induction n with d hd

rw add zero

rfl

rw add succ

rw hd

rfl

[1] It is important to learn to ask *interesting* questions. There is no precise way of defining what is meant by interesting. You can only learn this by doing. An interesting question comes typically in two parts. Part 1 (one or two sentences) sets the scene. Part 2 (one or two sentences) asks the question. A good question strikes the right balance between being specific and technical on the one hand and open ended on the other hand. A question that can be answered with yes/no is not an interesing question.

Level 2 - Succ Add

In this level we prove the theorem that **succ(a) + b = succ(a + b)**.

> **induction b with d hd**
>
> **repeat rw add zero**
>
> **rfl**
>
> **rw add succ**
>
> **rw hd**
>
> **rw add succ**
>
> **rfl**

Level 3 - Add Comm

In this level we prove the theorem that **a + b = b + a**.

> **induction b**
>
> **rw add zero**
>
> **rw zero add**
>
> **rfl**
>
> **rw succ add**
>
> **rw add succ**
>
> **rw nih**
>
> **rfl**

Level 4 - Add Assoc

In this level we prove the theorem that **(a + b) + c = a + (b + c)**.

> **induction b**
>
> **rw add zero**
>
> **rw zero add**
>
> **rfl**
>
> **rw add succ**
>
> **rw succ add**
>
> **rw nih**
>
> **rw add comm**
>
> **rw succ add**
>
> **rw add succ**
>
> **rfl**

Level 5 - Add Comm Right

In this level we prove the theorem that $\mathbf{(a + b) + c = (a + c) + b}$. I will show the mathematical assumptions I used when deciding my proof in Lean.

rw add assoc a b —— (RHS) By associativity rule, $\mathbf{a + c + b = a + (c + b)}$

rw add assoc —— (LHS) By associativity rule, $\mathbf{a + b + c = a + (b + c)}$

rw add comm b c —— (LHS) By commutative rule, $\mathbf{a + (b + c) = a + (c + b)}$

rfl —— Proves the goal $\mathbf{X = X}$

2.3 Week 3

Notes

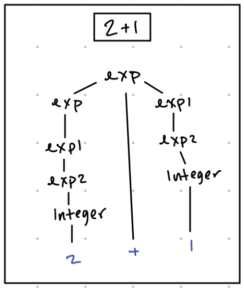
This week we talked a lot about context free grammars, especially within the context of parsing mathematical expressions. I really enjoyed learning more about CFGs, especially visualized, because it helped me to understand how CFGs are so important for programming languages. I also couldn't help but think how cool the visualized abstract syntax tree was. It makes me think more about how this sort of approach can be applied to other areas. I had unintentionally started designing my calculator using a similar paradigm, recursively splitting the expression into two small parts until each expression was complete. (I ended up switching approaches, but I thought it was cool anyway).

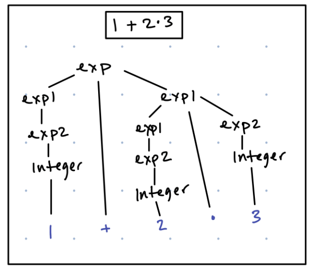
I was doing some additional research into ASTs, and was intrigued by the subsequent processes also required as part of compilation. Semantic analysis, for example. How many other steps are necessary for compilation? Does this process change for interpreted languages? I would be interested in learning more about the overhead costs of script compilation.

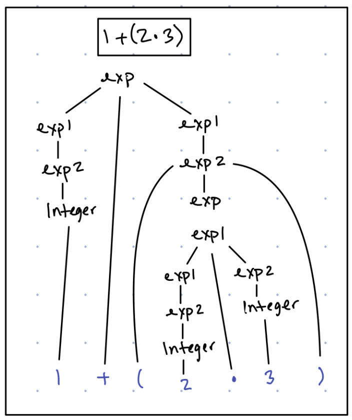
Question

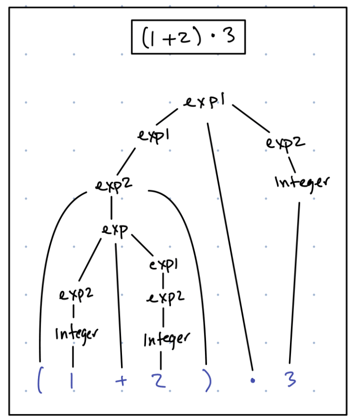
Last week we talked about context free grammars, especially within the context of mathematical syntax. Reading online, I learned about context-sensitive grammars. Can a CFG also be described as a CSG, or vice-versa? From what I could tell online, CSGs are invariably more complex than CFGs, so are there any actual applications of context sensitive grammars, and how is this reflected in its AST?

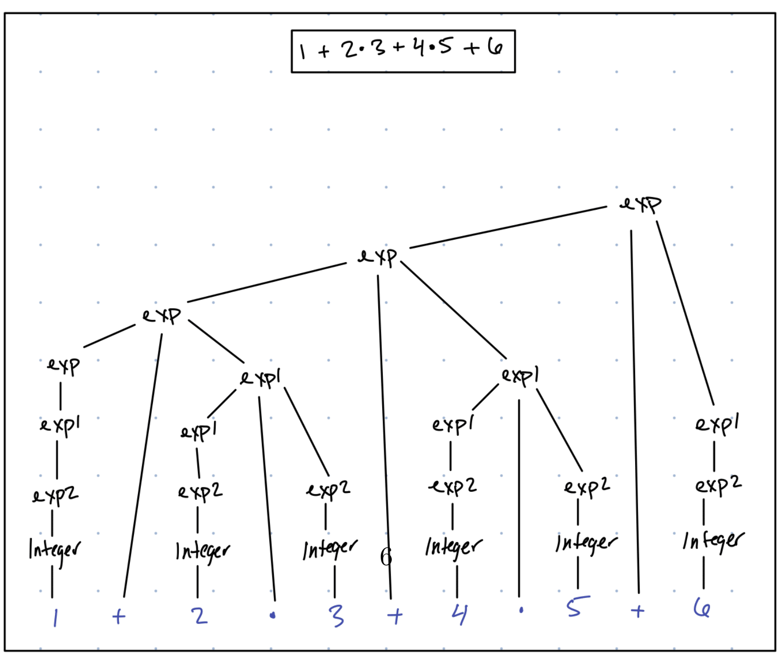
Homework


2+1
exp
exp1
exp2
Integer


1 + 2·3
exp
exp1
exp2
Integer


1+(2·3)
exp
exp1
exp2
Integer


(1+2)·3
exp
exp1
exp2
Integer


1 + 2·3 + 4·5 + 6
exp
exp1
exp2
Integer

2.4 Week 4

Notes

This week we discussed creating a calculator with CFGs and then about the next Lean world on logic. The logic world, similar to the math world, is about breaking down larger problems into manageable parts and tackling them one by one. If we can prove every infitessimal aspect of a larger problem, then we solve that large problem. I have really enjoyed working in the logic world so far because it is very similar to code design in game development. In game development, there are a number of components that control more context specific components, from the most high-level management context to the most basic mechanic.

Question

Similar to Lianal's question, why does the logic world in Lean allow for varying syntax, while the math Lean world did not? Is this reflected in real (written) math/logic proofs?

Homework

Level 1 - Exactly! It's in the premise

Assumptions: $(P : Prop)(todo\_list : P) : P := by$

(1) exact todo_list

Level 2 - And Introduction

Assumptions: $(PS : Prop)(p : P)(s : S) : P \wedge S := by$

(1) exact ⟨ p, s ⟩

Level 3 - The Have Tactic

Assumptions: $(AIOU : Prop)(a : A)(i : I)(o : O)(u : U) : (A \wedge I) \wedge O \wedge U := by$

(1) have ai := and_intro a i
(2) have ou := and_intro o u
(3) exact ⟨ ai, ou ⟩

Level 4 - And Elimination

Assumptions: $(PS : Prop)(vm : P \wedge S) : P := by$

(1) have p := vm.left
(2) exact p

Level 5 - And Elimination 2

Assumptions: $(PQ : Prop)(h : P \wedge Q) : Q := by$

(1) have q := h.right
(2) exact q

Level 6 - Mix and Match

Assumptions: $(AIOU : Prop)(h1 : A \wedge I)(h2 : O \wedge U) : A \wedge U := by$

(1) exact ⟨ h1.left, h2.right ⟩

Level 7 - More Elimination

Assumptions: $(CL : Prop)(h : (L \wedge (((L \wedge C) \wedge L) \wedge L \wedge L \wedge L)) \wedge (L \wedge L) \wedge L) : C := by$

(1) exact h.left.right.left.left.right

Level 8 - Rearranging Boxes

Assumptions: $(ACIOPSU : Prop)(h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge O) \wedge U) : A \wedge C \wedge P \wedge S := by$

(1) have c := h.right.right.left.left
(2) have psa := h.left
(3) have p := psa.left.left
(4) have s := psa.left.right
(5) have a := psa.right
(6) exact ⟨ a, c, p, s ⟩

2.5 Week 5

Notes

This week we discussed lambda higher order functions in the context of Lambda functions and had an interesting discussion about Curry Howard correspondance (currying) and languages. Lambda calculus functions are nameless, type-free functions. Within the context of lean, at least, this has been a difficult concept to grasp, because it is antithetical to the programming philosophy I have learned and practiced. However, the nature of lambda calculus is the same, but instead of relying on type-specific context, it relies on underlying logic within implicic type-deriving and preservation. I think this was confusing to me also because it throws explicit type safety out the window. On Thursday we discussed currying, α-Conversion, β-Reduction, (and more) and had an interesting discussion about languages, and meaning behind language.

Question

How do modern programming languages utilize (or avoid) lambda calculus in their syntax? In C#, linq utilizes higher order functions encapsulating (what appears to be) explicitly defined lambda calculus functions, for example "enum.Where(x ⇒ x.a > 10)". It also seems like SQL syntax follows currying of lambda calculus functions, for example "SELECT * FROM _ WHERE _ ORDER BY _", despite SQL not being a functional programming language.

Homework

Level 1 - Cake Delivery Service

Assumptions: $(PC : Prop)(p : P)(bakery\_service : P \rightarrow C) : C := by$

(1) exact (bakery_service p)

Level 2 - Identity

Assumptions: $(C : Prop) : C \to C := by$

(1) exact λ var $\mapsto$ var

Level 3 - Cake Form Swap

Assumptions: $(IS : Prop) : I \wedge S \to S \wedge I := by$

(1) exact λ h : I $\wedge$ S $\mapsto$ and_intro h.right h.left

Level 4 - And Elimination

Assumptions: $(CAS : Prop)(h1 : C \to A)(h2 : A \to S) : C \to S := by$

(1) exact λ c $\mapsto$ h2 (h1 c)

Level 5 - Riffin Snacks

Assumptions: $(PQRSTU : Prop)(p : P)(h1 : P \to Q)(h2 : Q \to R)(h3 : Q \to T)(h4 : S \to T)(h5 : T \to U) : U := by$

(1)exact h5 (h3 (h1 p))

Level 6 - and_imp

Assumptions: $(CDS : Prop)(h : C \wedge D \to S) : C \to D \to S := by$

(1) exact λ c d $\mapsto$ h (and_intro c d)

Level 7 - and_imp 2

Assumptions: $(CDS : Prop)(h : C \to D \to S) : C \wedge D \to S := by$

(1) exact λ $\langle$c, d$\rangle$ $\mapsto$ h c d

Level 8 - Distribute

Assumptions: $(CDS : Prop)(h : (S \to C) \wedge (S \to D)) : S \to C \wedge D := by$

(1) have $\langle$l, r$\rangle$:= h
(2) exact λ s $\mapsto$ $\langle$l s, r s$\rangle$

Level 9 - Uncertain Snacks

Assumptions: $(RS : Prop) : R \to (S \to R) \wedge (\neg S \to R) := by$

(1) exact λ r $\mapsto$ and_intro (λ _ $\mapsto$ r) λ _ $\mapsto$ r

2.6 Week 6

Notes

This week we dove into theory discussions and lectures on application of currying, more complex lambda calculus and church numerals. To be honest, it is hard to remember much notes-wise because the theory was so intense, but I can say with confidence that I was confused - and embraced it! To help with my understanding, I looked for resources online and found this paper by Helmut Brandl titled **Limits of Computability in Lambda Calculus** (`https://hbr.github.io/Lambda-Calculus/computability/text.html`[link to article]) which actually covered nicely what we have been learning about recently. In the preamble, Kurt Goedel's *Godel Numbering* is discussed, and I thought the paper gave some interesting historical context to a memorable moment in logic and maths history; "In his famous incompleteness theorem (1931) Goedel demonstrated that paradoxical statements can be injected into all formalisms which are powerful enough to express basic arithmetics." Anyway, I digress.

Question

Since lambda calculus is comprised of only abstraction and application, and cannot examine itself, how can infinitely recursing expressions like $(\lambda x.xx)(\lambda x.xx)$ terminate, and is that simply a non-issue? On a computer this would cause a stack overflow, but conceptually it follows the rules. Aside from physical computational constraints, are there other factors that do not permit translation from lambda calculus theory to implementation?

Homework

1) Reduce the following lambda term:

((\m.\n. m n) (\f.\x. f (f x)))
(\f.\x. f (f (f x)))

(1) ((\m.\n. m n) (\f.\x. f (f x))) (\f2.\x2. f2 (f2 (f2 x2)))

(2) (\n. (\f.\x. f (f x)) n) (\f2.\x2. f2 (f2 (f2 x2)))

(3) (\f.\x. f (f x)) (\f2.\x2. f2 (f2 (f2 x2)))

(4) \x. (\f2.\x2. f2 (f2 (f2 x2))) ((\f2.\x2. f2 (f2 (f2 x2))) x)

(5) \x.(\f2.\x2. f2 (f2 (f2 x2))) (\x2. x (x (x x2)))

(6) \x.\x2.(\x2. x (x (x x2))) ((\x2. x (x (x x2))) ((\x2. x (x (x x2))) x2))

(7) x\.\x2.(\x2. x (x (x x2))) ((\x2. x (x (x x2))) (x (x (x x2))))

(8) x\.\x2.(\x2. x (x (x x2))) (x (x (x (x (x (x x2))))))

(9) \x.\x2.x (x (x (x (x (x (x (x (x x2))))))))

2) Explain what function on natural numbers $(\backslash m.\backslash n.mn)$ implements:

This lambda expression implements similar behaviour to the identity function, but with church numerals. It returns the application of m on n.

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.