Q&A Session for Programming Languages Lecture 16

Session Number: 1204805728
Date: 2020-10-30
Starting time: 14:24

ANON - 14:23
Q: Are we allowed to use apply? for HW5
Priority: N/A
Ana L. Milanova - 14:54
A: Yes.

ANON - 14:27
Q: When are study resources and details about exam 2 going to be announced?
Priority: N/A
Ana L. Milanova - 14:55
A: On Tuesday, one week before the exam.

ANON - 14:28
Q: Can you please clarify which piece of the lambda expression is the parameter? Your example was unclear: lambda x.xy....you said x was the parameter. Is it the first x, the 2nd x, or the combination of x's?
Priority: N/A
Steven Haussmann - 14:31
A: The parameter of a lambda expression is the first name. For example: λx.y is an abstraction that has a parameter named "x". It completely ignores its argument.
Ana L. Milanova -
A: In (\lambda x. x y) we have an abstraction (this is just a term for function). Here x is the parameter, the parameter is always the one variable that follows the \lambda letter. (x y) is the body of the function. In this case, x is a function value, and we apply x on argument y. In this case, y is a free variable, i.e., non-local variable that is resolved in an outer scope.

ANON - 14:29
Q: Is the volumne on?
Priority: N/A
Konstantin Kuzmin - 14:30
A: We still have 1 minute. :)

ANON - 14:41
Q: Why is the first y in: lambda y. (y w) y considered bound when it is parenthesized?
Priority: N/A
Steven Haussmann - 14:46
A: The parentheses just indicate that we're applying y and w; since the y is within the body of the abstraction, it is bound

---

ANON - 14:45
Q: Sorry I'm a bit late to join, and forgive me if I ask the same question again, but I was wondering if you can clarify the page 11 last lecture? That is why we assign " λx.λy. x" to the meaning of "TRUE", but " λx.λy. y" to the meaning of "FALSE"?
Priority: N/A
Steven Haussmann - 14:47
A: I don't believe we've gotten to how we handle conditionals in the λ-calculus yet. In short, λx.λy.x takes two things and gives you the first one, whilst λx.λy.y takes two things and gives you the second one. This lets us pick between two things.
Steven Haussmann - 14:48
A: It has a similar structure to an if-then-else construct. You get the first part if the conditional is true, and you get the second part if the conditional is false.
Ana L. Milanova - 15:00
A: What Steven says. These lambda terms are known as the "Church booleans". Using these terms and their properties we can express if-then-else logic in the lambda calculus.

ANON - 14:47
Q: Regarding y being bound, so anything y in the body of lambda y will be bound./
Priority: N/A
Steven Haussmann - 14:48
A: Yes, although if there's another abstraction inside that also binds y, then any instances of y found inside of it will be bound by that inner abstraction, rather than the outer one.
Steven Haussmann - 14:49
A: No matter what, though, all instances of y inside *are* bound by something.

ANON - 14:48
Q: How does (lambda x . lambda y . lambda x . x * y) M A T get substituted?
Priority: N/A
Steven Haussmann - 14:52
A: This should result in T * A. The first application will combine (λx.λy.λx.x * y) with M; since the only x on the left side is

bound by the inner abstraction, the outer abstraction has no effect. y is then replaced with A, and x is then replaced with T.

ANON - 14:56
Q: what would (lambda (x) (x y)) evaluate to in scheme?
Priority: N/A
Steven Haussmann - 14:59
A: This would produce a function that takes an argument, then applies that argument onto a variable named y. If y does not exist when the function is executed, you will get an error.
Steven Haussmann - 14:59
A: (let ((y 1)) ((lambda (x) (x y)) number?)) returns true, for example.

ANON - 15:02
Q: (let ((y 1)) ((lambda (x) (x y)) number?)) returns true, for example. How can you apply number without an argument?
Priority: N/A
Steven Haussmann - 15:04
A: We're applying the lambda function to the number? predicate. The function accepts one argument, x. Thus, we evaluate (x y), where x is number? and y is 1. (number? 1) returns true.

ANON - 15:03
Q: does free occurrence refer to variables that are not defined in the inner scope?
Priority: N/A
Steven Haussmann - 15:07
A: See slide 5 again for the exact definition. A variable is free if there exists some path to that variable that doesn't go through an abstraction that binds it. So, if there are two instances, and only one is inside a binding abstraction, it's free

---

ANON - 15:05
Q: So we recursively substitute for Applications?
Priority: N/A
Ana L. Milanova - 15:06
A: Yes. We recurse in Application and in Abstraction.
Ana L. Milanova - 15:06
A: We recurse into the subexpressions Application, and also of Abstraction. E.g., in application E1 E2, we recurse into E1 then into E2.

ANON - 15:05
Q: Why is renaming important?

Priority: N/A

Ana L. Milanova - 15:07

A: Because if we do not rename (in some expressions) we will make a free variable bound, thus changing the meaning of the intended lambda term.

ANON - 15:06

Q: Oh I see. The lambda function is number? But then for this to return true, would we need let*? I'm trying to see if the Lambda Function is within the List of Bindings

Priority: N/A

Steven Haussmann - 15:08

A: No. The lambda function exists in an environment in which y=1. It accepts number? as an argument and applies it to y.

ANON - 15:07

Q: so beta-reduction rule is just defining substition?

Priority: N/A

Ana L. Milanova - 15:09

A: No, Beta reduction is function application. Each function application immediately triggers a substitution. Beta reduction and substitution are two different, but closely related, things.

ANON - 15:08

Q: In what cases do we skip renaming in substitution of an abstraction?

Priority: N/A

Steven Haussmann - 15:10

A: We stop renaming if we encounter another abstraction that binds the same name, since any occurences of the variable are obviously not free.

ANON - 15:09

Q: No. The lambda function exists in an environment in which y=1. It accepts number? as an argument and applies it to y.

Priority: N/A

ANON - 15:10

Q: Sorry I meant to say Oh I get it. Is it because the Lambda Function is the S-Expr? so it hads access to y =1?

Priority: N/A

ANON - 15:13

Q: would (Lx.E) M reduce to E{M/x] as the most reduced version?

Priority: N/A
Steven Haussmann - 15:14
A: You would have to actually perform the substitution - E[M/x] means "E, where all free occurrences of x are replaced with M"
Ana L. Milanova -
A: After you perform the substitution, the result may not necessarily be "the most reduced version". There could be other reducible expressions

---

ANON - 15:14
Q: so (Lambdaz. z z) (Lambda x. x) simplifies to (Lambda x. x) (Lambda x. x). Does this further simplify to just (Lambda x. x)? I feel like it does
Priority: N/A
Steven Haussmann - 15:15
A: Correct.

ANON - 15:25
Q: how do we know when there's a reducible expression?
Priority: N/A
Ana L. Milanova - 15:27
A: An application expression of the form (\x. E) M is a reducible expression. When there is an expression of this form, anywhere in our term, then we say that "there's a reducible expression".
Steven Haussmann - 15:30
A: To decide if there's *any* reducible expression, you have to traverse the entire expression, checking every place for an application of an abstraction with something. Weaker forms don't require complete traversal.

ANON - 15:29
Q: is reducible expression a specific type of application expression?
Priority: N/A
Ana L. Milanova - 15:30
A: Yes. It is an application expression E1 E2, where the first term, E1, is an abstraction.

ANON - 15:33
Q: how can we tell when something is an abstraction?
Priority: N/A
Ana L. Milanova - 15:33
A: Any term that starts with \lambda x. ... is an abstraction (or function definition).
Steven Haussmann - 15:34
A: There are three types of expressions: variables,

abstractions, and applications. Abstractions are written as λx.E, where x is the name of its parameter and E is its body

---

ANON - 15:33
Q: if something is in normal form, does that mean it will be in head normal form too?
Priority: N/A
Steven Haussmann - 15:34
A: Yes.

ANON - 15:35
Q: how come the example (Lz. z) is in normal form despite starting with \lambda
Priority: N/A
Steven Haussmann - 15:36
A: Simply having an abstraction does not mean there is a reducible expression. You can only reduce an application whose first expression is an abstraction. (λx.x a) is reducible; (λx.x) is not
Steven Haussmann - 15:37
A: An abstraction by itself is like a function that you've defined, but not used. It can't actually be run until you give it an argument.

ANON - 15:36
Q: NF implies HNF implies WHNF, right?
Priority: N/A
Ana L. Milanova - 15:37
A: Correct.

ANON - 15:37
Q: What is the definition of a combinator?
Priority: N/A
Ana L. Milanova - 15:38
A: A lambda term with no free variables. E.g., \x.x is a combinator. As another example, \x.\y.x is a combinator as well.
Steven Haussmann - 15:38
A: Importantly, this means that a combinator's meaning is completely independent of whatever encloses it.

ANON - 15:43
Q: why is S I reducible?
Priority: N/A
Ana L. Milanova - 15:44
A: Because S is an abstraction, i.e., a function value. S is just a shorthand notation for the large abstraction term.

ANON - 15:45
Q: So we substitute everything first, then evaluate?
Priority: N/A
Ana L. Milanova - 15:46
A: First we evaluate a redex, and that entails a substitution. The substitution then may result in more reducible expressions that we can evaluate.
Ana L. Milanova - 15:47
A: So, yes, we do a subsitution as part of the first evaluation step. Then we do another substitution as part of the second evaluation step, and so on.

ANON - 15:49
Q: So for the Last Question, we made the choice to do Normal Order Reduction and simply follow the Outer Reduction, when we could have evalute the arguments first, via Applicative Order?
Priority: N/A
Ana L. Milanova - 15:51
A: Yes, that is correct. (I will have to double check the example I did in the video, but I believe it was Normal Order.)

ANON - 15:50
Q: could we go over some of the reduction problems steps in more detail?
Priority: N/A
Ana L. Milanova - 15:52
A: We will do more problems in the beginning of class next time.

ANON - 15:50
Q: continue with the lecture is my vote
Priority: N/A
Ana L. Milanova - 15:53
A: Ok thanks!

ANON - 15:51
Q: Could you please provide practice problems and answers for reduction and normal form problems?
Priority: N/A
Ana L. Milanova - 15:55
A: We will provide additional problems in the practice problems sets for the test. I should have these done by the next class.

ANON - 15:56
Q: is one of these paths "better" than the other?
Priority: N/A
Ana L. Milanova - 15:58
A: We'll see later in lecture. Each one of the strategies has advantages and disadvantages.
Steven Haussmann - 15:58
A: Applicative order can get stuck in infinite loops. However, normal order can wind up doing a lot of extra work -- rather than evaluating a function's argument once, it might wind up evaluating many copies of that argument
Steven Haussmann - 15:58
A: oh, I've spoiled it (:
Ana L. Milanova - 15:59
A: Steven: no, not at all, thank you!

ANON - 15:56
Q: Is there a slide that talks about combinators and defines the commone ones?
Priority: N/A
Ana L. Milanova - 16:01
A: Slides 18 and 19 of lecture today defines the term "combinator" and also gave several examples of common combinators. I will try to find a good reference that lists a larger number of common combinators.

ANON - 16:00
Q: is there only two possible paths, or infinitely many?
Priority: N/A
Steven Haussmann - 16:02
A: There could be other reduction strategies. However, each one should always have exactly one choice for the next reduction.

ANON - 16:02
Q: How is (Ly.y) (Lz.z) the Left-Most? I agree it is Inner-Most, but I don't nderstand why that is Left-Most? Does it do Inner-Most first and then apply on the Left-Most?
Priority: N/A

ANON - 16:03
Q: So my Understanding is that we first search for the Most Inner/ Outer Redex and then find the Left-Most one? Not the other way around.
Priority: N/A
Ana L. Milanova - 16:04
A: Yes, that is correct. Innermost takes precedence. If there

are two expressions at the same "innermost" level, we chose the leftmost one.

ANON - 16:04
Q: Sorry, but she only defined combinator in speaking about it, she never formally introduced these terms. There are a lot of new terms and expressions being thrown around in these lectures and it sure would be a useful teaching tool to define them
Priority: N/A
        Ana L. Milanova - 16:06
        A: The term combinator is defined on slide 19: "An expression with no free variables is called combinator".
        Ana L. Milanova - 16:07
        A: If there is a term that is not defined, please let me know, and I will make a correction.

ANON - 16:08
Q: why don't they produce the same reuslt (slide 25)?
Priority: N/A
        Ana L. Milanova - 16:13
        A: Because applicative order reduction gets stuck into an infinite recursion, while normal order terminates.

ANON - 16:09
Q: We said that if Normal Form, then order does not matter, so should it not be te case that Applicative Order also should find the same Normal Form since order does not matter?
Priority: N/A
        Steven Haussmann - 16:11
        A: Applicative order can get stuck. There is still only one valid answer, but you might run forever, rather than finding the single unique answer.

ANON - 16:12
Q: I agree Applicative can get stuck. But iuf the Normal Form Exists, then it is unique which means order of computation does not matter which means Applicative should not fail in this case??
Priority: N/A
        Steven Haussmann - 16:13
        A: The existence of an answer does not mean you can actually reach it. If a program never terminates, then it never produces an answer at all.

ANON - 16:12
Q: Why would most programming languages use applicative order if it

isn't guaranteed to find the normal form? (or maybe I missed something?)
Priority: N/A
Steven Haussmann - 16:14
A: Pure normal form is very expensive -- for example, a map function would have to completely evaluate its function on every recursive step.
Ana L. Milanova - 16:15
A: Because normal order is more difficult to implement. It requires carrying unevaluated closures into the function value. Another reason is that applicative order is typically more efficient and we rarely have parameters that are not used in the function body.

ANON - 16:16
Q: was the S I I I from before normal order?
Priority: N/A
Ana L. Milanova - 16:19
A: I believe so. I will double check.

ANON - 16:18
Q: Was there a slide that talked about redicible expressions and how to know if it's reducible?
Priority: N/A
Ana L. Milanova - 16:19
A: Slide 10 defines the term redex (reducible expression).
Ana L. Milanova - 16:23
A: An application expression of the form (\x. E) M is a reducible expression. We can apply the function on the argument and reduce the original expression into a simpler expression. We will go over more examples next time to clarify these terms.

ANON - 16:25
Q: I'm still not sure why if the Church Theorem holds guarantees that order does not matter, why Applicative Order would not be guarranteed to succeed? Since it states the result of computation order does not matter
Priority: N/A
Ana L. Milanova -
A: The Church Rosser theorem proves confluence of the lambda calculus. The formal statement is this: ie e -*-> e1 and e -*-> e2 where these are some arbitrary sequences of reductions, then there exist e1 -*-> e3 and e2 -*-> e3. So this immediately gives us that if a normal form exists then it is unique (you can argue this by contradiction). And also, if a sequence of reductions reaches a normal form, then it reaches that unique normal form (that is what the informal statement means by "order doesn't matter").
A: But the theorem does not prevent some evaluation order from

"getting stuck" into an infinite recursion. We can have e -App Order-> e (This is our e -*-> e1 and we just have e1 = e) and e -Normal Order -> e3 where e3 is the normal form, as with our example. Confluence holds for this example as we have e -Normal Order-> e3, and e3 -*-> e3, the * means that we can have 0-reductions.

ANON - 16:25
Q: I thought the Q/A at the start of this lecture was really helpful! I still do find it difficult to apply these concepts to quickly after we've just learned them
Priority: N/A
Ana L. Milanova - 16:26
A: Thanks! We'll do another Q&A based on the questions we had this time around.

ANON - 16:41
Q: Also, I'm not sure about this. But a few LEctures back we discussed a count_sym Function in Scheme and never went over it. I would really appreicate going over that as well.
Priority: N/A
Ana L. Milanova -
A: This was just an example function to attempt as an exercise. You can implement any solution. We'll go over it if we have time.

ANON - 16:51
Q: No Office Hours Today, Just To Confirm?
Priority: N/A
Ana L. Milanova - 16:53
A: No, I have to cancel office hours today, I am sorry. I will also have to get back to the Q&A later than usual.