Answer To Assignment 2: Lambda Theory

VectorPikachu

2025 年 1 月 17 日

This week, we looked at the theory behind the untyped and simply-typed lambda calculi. In this assignment, you will do a deep dive into the computational power of the untyped lambda calculus. Rather than just considering language-level rules, you will write actual lambda calculus programs to perform numeric operations. You will also work through a few different changes and extensions to the lambda calculi.

1 Church Encoding

Part of understanding a computational model is knowing its expressiveness: what range of programs can be written in the model? For example, in the language of arithmetic, we know that every program must evaluate to a value, i.e. it can never loop. While most normal programming languages permit **partial** functions (functions that could return a result or loop forever), we've proven that arithmetic is a language of **total** functions (they always return a result).

Intuitively, this means that arithmetic is less expressive than more general programming languages. The set of programs expressible in arithmetic is a strict subset of those expressible in, say, Python or Java. What about the lambda calculus? It seems like such a simple language, but can we formally reason about its expressiveness? It turns out the lambda calculus is equivalent in expressiveness to a Turing machine—this result is called the Church-Turing thesis. Both of these computational models can express computable functions. Informally, a computable function is "a function on the natural numbers computable by a human being following an algorithm, ignoring resource limitations."

In the context of the lambda calculus, this idea will seem strange. If we only have variables and functions, how are supposed to express something as simple as 2+2? The key idea is that we have to provide a mapping from constructs in the lambda calculus to natural numbers, which allows us to interpret the result of a lambda calculus program as a number. This is called the Church encoding. In the remainder of this section, we'll work through a few exercises to understand how this encoding works.

1.1 Numbers(10%)

First, we'll look at how to interpret a lambda calculus program as a natural number. This construction derives from Peano numerals, which is essentially the "tally marks" approach to representing numbers that we discussed in the first lecture.

Here, Z is the number 0, $S(n)$ is the number $n+1$. For example:

- $Z = 0$
- $S(S(Z)) = 2$
- $S(S(S(S(S(Z))))) = 5$

A Church numeral for n is a lambda calculus function that takes two arguments f and x, and applies f to x a total of n times. For example:

- $\lambda\ f\ .\ \lambda\ x\ .\ x = Z = 0$
- $\lambda\ f\ .\ \lambda\ x\ .\ f\ (f\ x) = S(S(Z)) = 2$
- $\lambda\ f\ .\ \lambda\ x\ .\ f\ (f\ (f\ (f\ (f\ x)))) = S(S(S(S(S(Z))))) = 5$

Key idea: a function can be a number, and what's in the function defines exactly what number. It's all a matter of interpretation.
We can then define lambda calculus functions to emulate functions on numbers. For example, the successor operation adds one to the input number n:

$$S = \lambda\ n\ .\ \lambda\ f\ .\ \lambda\ x\ .\ f\ (n\ f\ x)$$

Why does this work? Let's walk through an example. If we evaluated $S(2)$, this would look like:

$$\begin{aligned} &(\lambda\ n\ .\ \lambda\ f\ .\ \lambda\ x\ .\ f\ (n\ f\ x))\ (\lambda\ f\ .\ \lambda\ x\ .\ f\ (f\ x)) \\ \mapsto &\lambda\ f\ .\ \lambda\ x\ .\ f\ ((\lambda\ f\ .\ \lambda\ x\ .\ f\ (f\ x))\ f\ x) && \text{[Substitute the 2 for } n\text{]} \end{aligned}$$

If this expression was the church numeral for 3, then it would apply the function f to x three times. Is that the case? Even though it's not technically part of the semantics, we can simplify inside parts of the number to answer this question.

$$\begin{aligned} &\lambda\ f\ .\ \lambda\ x\ .\ f\ ((\lambda\ f\ .\ \lambda\ x\ .\ f\ (f\ x))\ f\ x) \\ \mapsto &\lambda\ f\ .\ \lambda\ x\ .\ f\ (f\ (f\ x)) && \text{[Substitute inner } f \text{ and } x\text{]} \end{aligned}$$

From this, we can see that the resultant function does apply f three times to x, hence this is the church numeral for three, and our successor function appears to work.
Now let's mechanize this. In the `assign2/program directory`, we have provided a binary `lci_{platform}`, or lambda calculus interpreter, that executes an lambda calculus program and interprets it as a church numeral. Try this out:

```
$ ./lci_macos
> fun f -> fun x -> x
0
> z = fun f -> fun x -> x
> s = fun n -> fun f -> fun x -> f (n f x)
> (s (s z))
2
```

Note: this is a slightly modified version of the lambda calculus with persistent identifiers.
You can also use the interpreter to execute a standalone file.

```
$ cat test.lam
z = fun f -> fun x -> x
s = fun n -> fun f -> fun x -> f (n f x)
(s (s z))

$ ./lci_macos test.lam
Expression on line 3 evaluated to 2
```

Your first task is to define a multiply function. As per the Church encoding Wiki page, the addition and multiplication functions are usually defined as follows:

$$\mathsf{plus} = \lambda\ m\ .\ \lambda\ n\ .\ \lambda\ f\ .\ \lambda\ x\ .\ m\ f\ (n\ f\ x)$$
$$\mathsf{mult} = \lambda\ m\ .\ \lambda\ n\ .\ \lambda\ f\ .\ \lambda\ x\ .\ m\ (n\ f)\ x$$

Instead of this definition, your job is to provide an alternate implementation of mult that only uses $\{m, n, \mathsf{plus}, Z\}$, as well as parentheses. A few hints.

1. Fill in the template: $\mathsf{mult} = \lambda\ m\ .\ \lambda\ n\ .$ _____.
2. Remember that although m and n represent numbers, they are still functions that you can use however you like.
3. Use the interpreter to help you experiment!
4. A key idea here is **partial evaluation**. In this style of programming, if you give a function less than the expected number of arguments, it returns a function that waits on the remaining ones. For example, $\mathsf{plustwo} = \mathsf{plus}\ 2$ is a function that adds 2 to a number, e.g. $\mathsf{plustwo}\ 3 = 5$.

Put your answer in the `assign2/program/mult.lam` file.

Answer.

$$\mathsf{mult} = \lambda\ m\ .\ \lambda\ n\ .\ n\ (\mathsf{plus}\ m)\ Z.$$

最后结果是正确的:

```
z = fun f -> fun x -> x
s = fun n -> fun f -> fun x -> f (n f x)
plus = fun m -> fun n -> m s n
mult = fun m -> fun n -> n (plus m) z

mult z z
mult (s z) (s z)
mult (s z) (s (s z))
mult (s z) (s (s (s z)))
```

```
mult (s (s z)) (s (s (s z)))
mult z (s (s (s z)))
mult (s (s z)) (mult (s (s z)) (s (s z)))

ten = (s (s (s (s (s (s (s (s (s z))))))))))
mult ten ten
```

```
Value on line 5: 0
Value on line 6: 1
Value on line 7: 2
Value on line 8: 3
Value on line 9: 6
Value on line 10: 0
Value on line 11: 8
Value on line 13: 100
```

1.2 Booleans (10%)

To introduce conditional logic, i.e. ifs/thens/comparisons, we need to use the Church encoding for booleans. Here, the idea is that `true` and `false` are functions which select a particular branch of two possibilities.

$$\mathsf{true} = \lambda\ a\ .\ \lambda\ b\ .\ a$$
$$\mathsf{false} = \lambda\ a\ .\ \lambda\ b\ .\ b$$

For example, $\mathsf{true}\ Z\ (S\ Z) \mapsto Z$. From this, we can define our essential conditional operators:

$$\mathsf{not} = \lambda\ p\ .\ \lambda\ a\ .\ \lambda\ b\ .\ p\ b\ a$$
$$\mathsf{if} = \lambda\ p\ .\ \lambda\ a\ .\ \lambda\ b\ .\ p\ a\ b$$

Again, see the Wiki page for more explanation and examples. In this section, your challenge is to define a function even that takes as input a number, and returns true if it's even, and false otherwise. You can only use the input number and the $\{\mathsf{true}, \mathsf{false}, \mathsf{not}, \mathsf{if}\}$ constructs (no recursion!). Some hints:

1. Fill in the template: $\mathsf{even} = \lambda\ n\ .$ _____.
2. We can inductively define the even function over the natural numbers as: $\mathsf{even}\ Z = \mathsf{true}$ | $\mathsf{even}\ S(n) = \mathsf{not}\ (\mathsf{even}\ n)$.

Put your answer in the `assign2/program/even.lam` file.

Answer.

首先给出答案:

$$\mathsf{even} = \lambda\ n\ .\ n\ \mathsf{not}\ \mathsf{true}$$

最后even.lam里的结果是对的:

```
z = fun f -> fun x -> x
s = fun n -> fun f -> fun x -> f (n f x)
true = fun a -> fun b -> a
false = fun a -> fun b -> b
not = fun p -> fun a -> fun b -> p b a
even = fun n -> n not true

booltonum = fun b -> b (s z) z

booltonum (even z)
booltonum (even (s z))
booltonum (even (s (s z)))
booltonum (even (s (s (s z))))

booltonum (even (s (s (s (s (s (s (s (s z)))))))))
```

```
Value on line 8: 1
Value on line 9: 0
Value on line 10: 1
Value on line 11: 0
Value on line 12: 1
```

1.3 Recursion (10%)

A last critical piece missing from our computational framework is the idea of recursion. In the basic formulation given, you can't trivially recurse within a function. When declaring a function, you can't get access to the identifier binding the function. For example, when defining a recursive sum in Javascript, we can usually write:

```
function sum(n) {
    if (n == 0) { return 0; }
    else { return sum(n - 1) + n; }
}
```

Where we are allowed to reference sum inside of its own definition. But in the lambda calculus, if we try this idea:

$$\mathsf{sum} = \lambda\ n\ .\ \mathsf{if}\ (\mathsf{iszero}\ n)\ Z\ (\mathsf{plus}\ n\ (\mathsf{sum}\ (\mathsf{pred}\ n)))$$

Here, iszero returns true if the input is zero and false otherwise. $\mathsf{pred}\, n$ returns $n - 1$.
Then this is an invalid expression, because sum isn't bound in the function. Instead, we can use the amazing magics of the **fixpoint combinator** (or more famously, the "Y combinator"). The

fixpoint combinator looks like this:

$$\mathsf{fix} = \lambda\ f\ .\ (\lambda\ x\ .\ f\ (x\ x))\ (\lambda\ x\ .\ f\ (x\ x))$$

The fix function has the curious property of a fixpoint, which is that $\mathsf{fix}\ f = f\ (\mathsf{fix}\ f)$ for all f. See an extended example of this on the Wiki page.
Intuitively, the way we can use the fixpoint combinator is to provide a function with a handle to itself—this is the idea of self-reference. For example, we can define an infinite sum:

$$\begin{aligned}
&\mathsf{fix}\ (\lambda\ n\ .\ n+1)\\
\mapsto&(\lambda\ x\ .\ (\lambda\ n\ .\ n+1)\ (x\ x))\ (\lambda\ x\ .\ (\lambda\ n\ .\ n+1)\ (x\ x))\\
\mapsto&(\lambda\ n\ .\ n+1)\ ((\lambda\ x\ .\ (\lambda\ n\ .\ n+1)\ (x\ x))\ (\lambda\ x\ .\ (\lambda\ n\ .\ n+1)\ (x\ x)))\\
=&(\lambda\ n\ .\ n+1)\ (\mathsf{fix}\ (\lambda\ n\ .\ n+1))
\end{aligned}$$

More practically, we can define recursive functions that terminate using conditions like in the Javascript example above. Your challenge in this section is to use the fix combinator to create a recursive sum function that adds the numbers from 1 to n. You can use any of the constructs defined in the above sections or on the linked Wikipedia pages (except don't use the closed form!). Put your answer in `assign2/program/sum.lam`.

If you wanted to, how would you complete the proof of the equivalence of the lambda calculus and Turing machines? Once you have numbers, booleans, and a fixpoint, it's only a short jump to simulating a Turing machine inside the lambda calculus. You would need a notion of lists and tuples (and a way of getting input), and then you can define a function:

```
step : (number /*state*/, list /*tape*/) -> number /*input*/ -> (number /*state*/, list
↪  /*tape*/)
s0 = pair 0 empty
tm = fix (fun tm -> fun s -> tm (step s (get_input ())))
```

Out of respect for your time, I'll leave this as an optional exercise for the reader.

Answer.
生成递归函数的办法:

1. 构造一个生成器函数: $F = \lambda\ \mathsf{self}\ .\ f$.
2. 把 F 传入 fix, 得到递归版本: $F_{\mathsf{rec}} = \mathsf{fix}\ F$.

那么我们这里的答案就是:

$$\mathsf{sum} = \mathsf{fix}\ (\lambda\ \mathsf{self}\ .\ \lambda\ n\ .\ \mathsf{if}\ (\mathsf{iszero}\ n)\ Z\ (\mathsf{plus}\ n\ (\mathsf{self}\ (\mathsf{pred}\ n))))$$

同时我们发现 iszero 还没有被定义, 我们定义如下:

$$\mathsf{iszero} = \lambda\ n\ .\ n\ (\lambda\ x\ .\ \mathsf{false})\ \mathsf{true}$$

我们在`sum.lam`里的结果是对的:

```
z = fun f -> fun x -> x
s = fun n -> fun f -> fun x -> f (n f x)
pred = fun n -> fun f -> fun x -> n (fun g -> fun h -> h (g f)) (fun u -> x) (fun u -> u)

plus = fun m -> fun n -> n s m

true = fun a -> fun b -> a
false = fun a -> fun b -> b
if = fun p -> fun a -> fun b -> p a b

fix = fun f -> (fun x -> f (x x)) (fun x -> f (x x))

iszero = fun n -> n (fun x -> false) true
sum = fix (fun self -> fun n -> if (iszero n) z (plus n (self (pred n))))

sum z
sum (s z)
sum (s (s z))
sum (s (s (s (s (s z)))))
sum (s (s (s (s (s (s (s (s (s z))))))))))
```

```
Value on line 11: 0
Value on line 12: 1
Value on line 13: 3
Value on line 14: 15
Value on line 15: 55
```

现在我们来说明 Lambda calculus 和 Turing machine 的对应关系.
首先, 图灵机的定义是: 一个图灵机是一个七元有序组 $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, 其中 Q, Σ, Γ 都是有限集合, 且满足:

1. Q 是非空有穷状态集合;
2. Σ 是非空有穷输入字母表, 其中不包含特殊的空白符 $\square$;
3. Γ 是非空有穷带字母表且 $\Sigma \subset \Gamma$; $\square \in \Gamma - \Sigma$ 为空白符, 也是唯一允许出现无限次的字符;
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, -\}$ 为状态转移函数, 其中 L, R, $-$ 分别表示向左, 向右移动和不移动;
5. $q_0 \in Q$ 是起始状态;
6. $q_{\text{accept}} \in Q$ 和 $q_{\text{reject}} \in Q$ 是接受状态和拒绝状态. 且 $q_{\text{accept}} \neq q_{\text{reject}}$.

我们考察上面的定义并且给出对应关系:

1. Q 是非空有穷状态集合, 对应为 numbers 的集合.

2. Σ 是非空有穷输入字母表, 其中不包含特殊的空白符 $\square$, 也可以被对应为不含 Z 的 numbers 的集合.
3. Γ 含有 $\square$, 那么就加上 Z.
4. δ 就是接受一个当前的状态和输入, 返回一个下一个状态, 输入, 以及一个移动方向. 对应为 step 函数.
5. q_0 就是 $s0$ 对应的状态.

那么之前定义的`tm`就是一个不停地接受当前 tape 的 state, 然后结合上输入, 然后返回下一个 state, 输入, 以及移动方向的函数.

2 Dynamic scoping (30%)

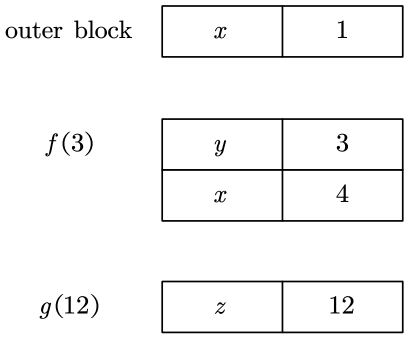
Credit to Stephan Boyer for the inspiration.

Lexical vs. dynamic scoping

Recall from class that scope is the central idea in programming languages of: given a variable, what binding does it refer to? For example, consider this program (in Javascript):

```
let x = 1;
function g(z) { return x + z; }
function f(y) {
  let x = y + 1;
  return g(y * x);
}
console.log(f(3));
```

When executing the inner g function, it has the following call stack:



Take a second to walk through the code and verify this diagram matches your expectations. When executing `g`, the name resolution question here is: in the expression `x + z` what does `x` refer to? There are two options: the `x` defined in the top-level scope, and the `x` defined in the function `f`. The first option, picking `x = 1`, is called **lexical scoping**. The intuition behind lexical scoping is that the program structure defines a series of "scopes" that we can infer just by reading the program, without actually running the code.

```
// The {N, ...} syntax indicates which stack of scopes are "active" a given
// point in the program.

// BEGIN SCOPE 1 -- {1}
let x = 1;
function g(z) {
    // BEGIN SCOPE 2 -- {1, 2}
    return x + z;
    // END SCOPE 2
}
function f(y) {
  // BEGIN SCOPE -- {1, 3}
  let x = y + 1;
  return g(y * x);
  // END SCOPE
}
console.log(f(3));
// END SCOPE 1
```

The nested structure of our program implies that at any line, we have a stack of scopes as annotated in the program above. To resolve a name, we walk the stack and search for the closest declaration of our desired variable. In the example, because scope 3 is not in the stack of scope 2, `x + z` resolves to the binding of `x` in scope 1, not scope 3.

Again, we call this "lexical" scoping because the scope stack is determined statically from the program source code (i.e. before the program is executed). The alternative approach, the one that resolves to `x = 4`, is called dynamic scoping. Recall our call stack above:

outer block	x	1
$f(3)$	y	3
	x	4
$g(12)$	z	12

As you may have inferred at this point, dynamic scoping looks at the runtime call stack to find the most recent declaration of `x` and uses that value. Here, the binding `x = 4` in `f` is more recent than the binding `x = 1` in the top-level scope.

In both cases, dynamic or lexical, the core algorithm is the same: given a stack of scopes, walk up the stack to find the most recent name declaration. The only difference is whether the the scope stack is determined before or during the program's execution.

Scoping in the lambda calculus

Let's now contextualize these ideas in the lambda calculus. The classical presentation of the lambda calculus is always lexically scoped, i.e. a variable always refers to the closest lambda with the same argument variable in the syntax tree. However, it's possible to tweak the evaluation rules and implement dynamic scoping in the lambda calculus. For example, consider the expression:

$$(\lambda\ x\ .\ (\lambda\ x\ .\ \lambda\ \_\ .\ x)\ L\ *)\ D$$

Conventions: as a shorthand, we're using the underscore for a variable that doesn't matter, and the asterisk for a value whose specific value does not matter (it could be anything and the example does not change). L and D are both values.
Under normal lexical scoping rules, this evaluates to:

$$\begin{aligned}
&(\lambda\ x\ .\ (\lambda\ x\ .\ \lambda\ _\ .\ x)\ L\ *)\ D \\
\mapsto &(\lambda\ x\ .\ \lambda\ _\ .\ x)\ L\ * && \text{[Inner } x \text{ shadows outer } x\text{]} \\
\mapsto &(\lambda\ _\ .\ L)\ * && \text{[Substitute } L \text{ for innermost } x\text{]} \\
\mapsto &L && \text{[Throw away dummy value]}
\end{aligned}$$

However, under a dynamic scoping system, the exact same example would evaluate to a different answer. Here's the trace under such a scoping system:

$$\begin{aligned}
&(\lambda\ x\ .\ (\lambda\ x\ .\ \lambda\ _\ .\ x)\ L\ *)\ D \\
\mapsto &(\lambda\ x\ .\ (\lambda\ _\ .\ x)\ *)\ D && \text{[Step 1]} \\
\mapsto &(\lambda\ x\ .\ (\lambda\ _\ .\ D)\ *)\ D && \text{[Step 2]} \\
\mapsto &(\lambda\ x\ .\ D)\ D && \text{[Step 3]} \\
\mapsto &D && \text{[Step 4]}
\end{aligned}$$

This execution trace should look strange to you. We're evaluating the inside of a function without substituting the variable! Let's look at the operational semantics to better understand this idea.

$$\frac{}{\lambda\ x\ .\ e\ \mathsf{val}}\text{(D-Lam)} \quad \frac{x \to e \in \Gamma}{\Gamma \vdash x \mapsto e}\text{(D-Var)} \quad \frac{\Gamma \vdash e_{\mathsf{lam}} \mapsto e'_{\mathsf{lam}}}{\Gamma \vdash e_{\mathsf{lam}}\ e_{\mathsf{arg}} \mapsto e'_{\mathsf{lam}}\ e_{\mathsf{arg}}}\text{(D-App-Lam)}$$

$$\frac{\Gamma, x \to e_{\mathsf{arg}} \vdash e_{\mathsf{body}} \mapsto e'_{\mathsf{body}}}{\Gamma \vdash (\lambda\ x\ .\ e_{\mathsf{body}})\ e_{\mathsf{arg}} \mapsto (\lambda\ x\ .\ e'_{\mathsf{body}})\ e_{\mathsf{arg}}}\text{(D-App-Body)} \quad \frac{e_{\mathsf{body}}\ \mathsf{val}}{\Gamma \vdash (\lambda\ x\ .\ e_{\mathsf{body}})\ e_{\mathsf{arg}} \mapsto e_{\mathsf{body}}}\text{(D-App-Done)}$$

Whereas in lecture we used a proof context in the typing rules, here we use a proof context for our operational semantics. The proof context is a set of variable mappings, similar to our runtime stack in the Javascript example. The syntax $\Gamma, x \to e_2$ means "add $x \to e_2$" to the context Γ. Like a normal dictionary, a new mapping can overwrite a previous one, for example:

$$\{y \to e_1\}, x \to e_2 = \{y \to e_1, x \to e_2\}$$
$$\{x \to e_1\}, x \to e_2 = \{x \to e_2\}$$

The crux of the construction is in the D-App-Body rule. To evaluate a function, we step the body e_1 while replacing $x \to e_2$ nto the proof context. This allows any expression within e_1 to use x,

even if it wasn't in its lexical scope. Leaving e_2 in the syntax while we step the function body is essentially using the syntax to maintain a runtime stack, which we reconstruct in our proof context on every small step.

For another clarifying example, here's a valid expression:

$$(\lambda\ f\ .\ (\lambda\ x\ .\ f\ \_)\ D)\ (\lambda\ \_\ .\ x)$$

Even though the x is not in the lexical scope of the function f, this will still evaluate to D because x is bound while f is being executed.

Stop. Re-read the above two sections and ensure that you understand what dynamic scoping means, and why these semantics implement it. Most of your time in this part of the assignment will likely be spent understanding the background, not performing the task.

2.1 Evaluation proof (15%)

Using the above semantics, we can produce a formal proof to justify each step in the provided example trace. Below, we have provided you with the first step in the proof—your task is to fill in the proof for the remaining three steps.

Step 1:
$$\dfrac{\dfrac{\dfrac{\dfrac{}{\lambda\ \_\ .\ x\ \mathsf{val}}\text{(D-Lam)}}{\{x \to D\} \vdash (\lambda\ x\ .\ \lambda\ \_\ .\ x)\ L \mapsto \lambda\ \_\ .\ x}\text{(D-App-Done)}}{\{x \to D\} \vdash (\lambda\ x\ .\ \lambda\ \_\ .\ x)\ L\ * \mapsto (\lambda\ \_\ .\ x)\ *}\text{(D-App-Lam)}}{\emptyset \vdash (\lambda\ x\ .\ (\lambda\ x\ .\ \lambda\ \_\ .\ x)\ L\ *)\ D \mapsto (\lambda\ x\ .\ (\lambda\ \_\ .\ x)\ *)\ D}\text{(D-App-Body)}$$

Answer.

Step 2:
$$\dfrac{\dfrac{\dfrac{x \to D \in \{x \to D, \_ \to *\}}{\{x \to D, \_ \to *\} \vdash x \mapsto D}\text{(D-Var)}}{\{x \to D\} \vdash (\lambda\ \_\ .\ x)\ * \mapsto (\lambda\ \_\ .\ D)\ *}\text{(D-App-Body)}}{\emptyset \vdash (\lambda\ x\ .\ (\lambda\ \_\ .\ x)\ *)\ D \mapsto (\lambda\ x\ .\ (\lambda\ \_\ .\ D)\ *)\ D}\text{(D-App-Body)}$$

Step 3:
$$\dfrac{\dfrac{D\ \mathsf{val}}{\{x \to D\} \vdash (\lambda\ \_\ .\ D)\ * \mapsto D}\text{(D-App-Done)}}{\emptyset \vdash (\lambda\ x\ .\ (\lambda\ \_\ .\ D)\ *)\ D \mapsto (\lambda\ x\ .\ D)\ D}\text{(D-App-Body)}$$

Step 4:
$$\dfrac{D\ \mathsf{val}}{\emptyset \vdash (\lambda\ x\ .\ D)\ D \mapsto D}\text{(D-App-Done)}$$

这其中 D val 来自题目已知条件.

2.2 New semantics (15%)

Your final challenge is to extend the dynamically scoped lambda calculus with a new construct, the let statement. Let statements are like functions in that they bind variables for use in an expression.

For example:

$$(\mathsf{let}\ x = 1\ \mathsf{in}\ x + x) \overset{*}{\mapsto} \mathsf{let}\ x = 1\ \mathsf{in}\ 1 + 1 \overset{*}{\mapsto} 2$$

We could rewrite the initial example as:

$$\mathsf{let}\ x = D\ \mathsf{in}\ (\mathsf{let}\ x = L\ \mathsf{in}\ \lambda\ \_\ .\ x)\ * \overset{*}{\mapsto} D$$

More generally, we extend our grammar with the following construct:

$$\begin{aligned} e ::= \quad & \ldots \\ & \mathsf{let}\ x = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}} \end{aligned}$$

Provide the operational semantics for the let statement that preserves dynamic scoping. The reference solution contains two inference rules. Put your answer in part 2 of the written solution.

Answer.

应该有两条 inference rule, D-Let-Body 和 D-Let-Done.

$$\frac{\Gamma, x \to e_{\mathsf{var}} \vdash e_{\mathsf{body}} \mapsto e'_{\mathsf{body}}}{\Gamma \vdash \mathsf{let}\ x = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}} \mapsto \mathsf{let}\ x = e_{\mathsf{var}}\ \mathsf{in}\ e'_{\mathsf{body}}}(\text{D-Let-Body})$$

$$\frac{e_{\mathsf{body}}\ \mathsf{val}}{\Gamma \vdash \mathsf{let}\ x = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}} \mapsto e_{\mathsf{body}}}(\text{D-Let-Done})$$

3 Typed language extensions (40%)

Eager to gain the approval of the functional programming community, Will has submitted two potential extensions to the simply-typed lambda calculus. For each extension, he wrote down the proposed statics and dynamics. Unfortunately, his eagerness led him astray, and only one of these extensions is type safe—the other **violates** at least one of progress and preservation.

Your task is to identify which extension violates these theorems and provide a counterexample, then for the other extension provide a proof of both progress and preservation for the given rules. Your counter example should be rigorous, meaning you should do more than just provide an example expression that violates the theorems. You must demonstrate why it's a violation.

First, recall from lecture the definitions of progress and preservation.

Progress: if $e : \tau$, then either e val or there exists an e' such that $e \mapsto e'$.

Preservation: if $e : \tau$ and $e \mapsto e'$ then $e' : \tau$.

The first extension adds `let` statements to the language, which emulates traditional (immutable) variable binding in Turing languages.

$$\frac{\Gamma, x : \tau_{\mathsf{var}} \vdash e_{\mathsf{body}} : \tau_{\mathsf{body}}}{\Gamma \vdash (\mathsf{let}\ x : \tau_{\mathsf{var}} = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}}) : \tau_{\mathsf{body}}}(\text{T-Let}) \qquad \frac{}{\mathsf{let}\ x : \tau = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}} \mapsto [x \to e_{\mathsf{var}}]\ e_{\mathsf{body}}}(\text{D-Let})$$

For example, this would allow us to write:

$$\mathsf{let}\ x : \mathsf{num} = 3\ \mathsf{in}\ x + 2 \mapsto 3 + 2 \mapsto 5$$

The second extension adds a recursor (`rec`) to the language. A recursor is like a for loop (or more accurately a "fold") that runs an expression from 0 to n:

$$\frac{\Gamma \vdash e_{\mathsf{arg}} : \mathsf{num} \quad \Gamma \vdash e_{\mathsf{base}} : \tau \quad \Gamma, x_{\mathsf{num}} : \mathsf{num}, x_{\mathsf{acc}} : \tau \vdash e_{\mathsf{acc}} : \tau}{\Gamma \vdash \mathsf{rec}(e_{\mathsf{base}};\ x_{\mathsf{num}}.x_{\mathsf{acc}}.e_{\mathsf{acc}})(e_{\mathsf{arg}}) : \tau}(\text{T-Rec})$$

$$\frac{e \mapsto e'}{\mathsf{rec}(e_{\mathsf{base}};\ x_{\mathsf{num}}.x_{\mathsf{acc}}.e_{\mathsf{acc}})(e) \mapsto \mathsf{rec}(e_{\mathsf{base}};\ x_{\mathsf{num}}.x_{\mathsf{acc}}.e_{\mathsf{acc}})(e')}(\text{D-Rec-Step})$$

$$\frac{}{\mathsf{rec}(e_{\mathsf{base}};\ x_{\mathsf{num}}.x_{\mathsf{acc}}.e_{\mathsf{acc}})(0) \mapsto e_{\mathsf{base}}}(\text{D-Rec-Base})$$

$$\frac{n > 0}{\mathsf{rec}(e_{\mathsf{base}};\ x_{\mathsf{num}}.x_{\mathsf{acc}}.e_{\mathsf{acc}})(n) \mapsto [x_{\mathsf{num}} \to n, x_{\mathsf{acc}} \to \mathsf{rec}(e_{\mathsf{base}};\ x_{\mathsf{num}}.x_{\mathsf{acc}}.e_{\mathsf{acc}})(n-1)]\ e_{\mathsf{acc}}}(\text{D-Rec-Dec})$$

Here's a few examples of using a recursor:

$$\mathsf{rec}(0;\ x.y.x + y)(n) = 0 + 1 + ... + n$$
$$\mathsf{rec}(1;\ x.y.x * y)(n) = 1 * 1 * 2 * ... * n$$

Put your answer in part 3 of the written solution.

Answer.

`rec`扩展违反了保持性, 比如对于下面的这个例子:

$$\mathsf{rec}(\mathsf{true};\ x.y.x + y)(1)$$

首先根据操作语义, 这个表达式可以 step:

$$\mathsf{rec}(\mathsf{true};\ x.y.x + y)(1) \mapsto [x \to 1, y \to \mathsf{rec}(\mathsf{true};\ x.y.x + y)(0)]\, x + y$$

而我们有:

$$\mathsf{rec}(\mathsf{true};\ x.y.x + y)(0) \mapsto \mathsf{true}$$

这样就会使得:

$$[x \to 1, y \to \mathsf{rec}(\mathsf{true};\ x.y.x + y)(0)]\, x + y \mapsto [x \to 1, y \to \mathsf{true}]\, x + y$$

这就是保持性被违反了.

我们首先说明`let`扩展的 Progress:

当 $e : \tau$ 是应用 T-Let 的时候, 我们发现总是可以使用 D-Let 来 step.

接着我们说明 Preservation:

当 $e : \tau$ 是应用 T-Let 的时候, $e \mapsto e'$ 只能使用 D-Let, 那么 $e' = [x \to e_{\mathsf{var}}]\, e_{\mathsf{body}}$, 而这根据对 T-Let 使用 invert lemma 就知道 $\tau = \tau_{\mathsf{body}}$, $e' : \tau_{\mathsf{body}}$. 那么 preservation 就得证了.