Lecture 6: Functions and Recursion

v2 (31st January 2025)

Programming Languages (H)

Simon Fowler & Michele Sevegnani

Semester 2, 2024/2025

sli.do 1275413

University of Glasgow

sli.do 1275413

Overview

- **Last time**
  - Variables, scope, and let-binding
- **This lecture:** Functions and Recursion
  - Anonymous functions
  - L_{Lam} syntax and semantics
  - Variable capture and Capture-avoiding substitution
  - Recursive functions

sli.do 1275413

The Need for Functions

- Let-bindings are useful as they give us a way of **abbreviating** expressions
- However, they do not let us write **functions**: expressions which can make use of **arguments**
- We can't, for example, use L_{Let} to describe an 'add5' function that adds 5 to a given number

```
let add5 = ??? + 5 in add5 10
```

sli.do 1275413

Functions

```
doubleAndAdd4(x) =
  2 * x + 4

id(x) = x

square(x) = x * x
```

- Remember from AF2: a mathematical function from set X to set Y is a mapping from every element of X to an element in Y
- In the context of programming languages, a function is any expression that can take an **argument** and produce some **result** based on that argument.
- On the left we can see some mathematical functions: in each of them the function name has a **parameter** and free occurrences of the parameter in the body

sli.do 1275413

Function bindings

```
let fun square x =
  x * x
in

square 5 + square 10
```

```
let fun square x = x * x in
let fun double x = x * 2 in
let fun doubleAndSquare x =
  square (double x)
in
doubleAndSquare 100
```

- As a first attempt, let us extend our let binding from L_{Let}
- The **`let fun`** `f x = M` **`in`** `N` construct defines a function `f` with parameter `x` and body `M`, allowing it to be used in body `N`
- We also need a construct for **application**, in this case of the form `f M`

sli.do 1275413

Anonymous Functions (Lambdas)

There's a simpler, more primitive way of representing functions: **anonymous** (or **lambda**) functions

$$\lambda \underbrace{n}_{\textbf{Parameter}}.\ \underbrace{n + 5}_{\textbf{Body}}$$

We can then **apply** a function, meaning we replace the free occurrences of the parameter with an argument before evaluating

$$(\lambda n.\ n + 5)\ 10 \Downarrow 15$$

sli.do 1275413

Multi-Argument Functions

Lambda expressions only have a **single** parameter, but we can define functions with multiple parameters by **nesting** multiple functions

```
((λn1. (λn2. n1 + n2)) 5) 10
↓
(λn2. 5 + n2) 10
↓
5 + 10
↓
15
```

This is known as **currying**, after the logician Haskell Curry

sli.do 1275413

Let / Let-fun are Syntactic Sugar!

```
let fun f x = M in N
         ⇔
let f = (λx. M) in N
```

```
let x = M in N
      ⇔
  (λx. N) M
```

- In fact, as soon as we have anonymous functions and application, we can **encode** both **`let fun`** and **`let`**
  - They are much easier to write, though, so it's worth keeping them in the source language as **syntactic sugar**
  - We don't, however, need to write explicit reduction / substitution rules for them
- We would first need to desugar **`let fun`** into a **`let`** binding, and then desugar the **`let`** binding into lambda expressions and function applications

sli.do 1275413

Desugaring Example

```
let fun add5 x = x + 5 in add5 10
```

(desugars to)

```
let add5 = λx. x + 5 in add5 10
```

(desugars to)

```
(λadd5. add5 10) (λx. x + 5)
```

sli.do 1275413

L_{Lam} Abstract Syntax

L_{Lam} Abstract Syntax

```
Integers n
Variables x, y, z
Operators ⊙   ::= + | - | *  | /
Values V, W   ::= n | λx . M
Terms L, M, N ::= x | n
                | λx . M
                | M N
                | L ⊙ M
```

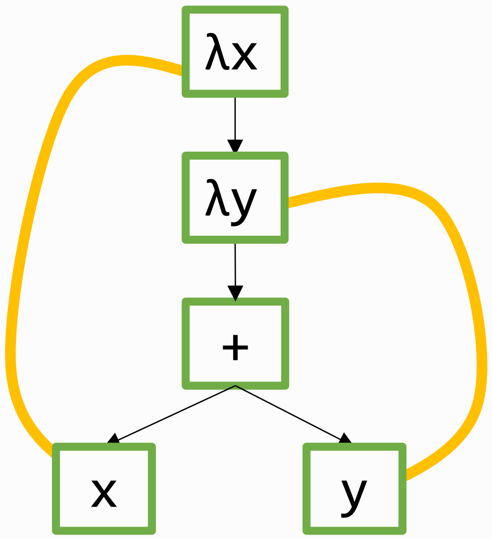
- We extend the language again with variables, and also anonymous functions `λx. M` and function application `M N`
- We need not consider **`let`** or **`let fun`** expressions in the abstract syntax; we can assume that they have already been encoded
- We will write `\x. M` rather than `λx. M` in code

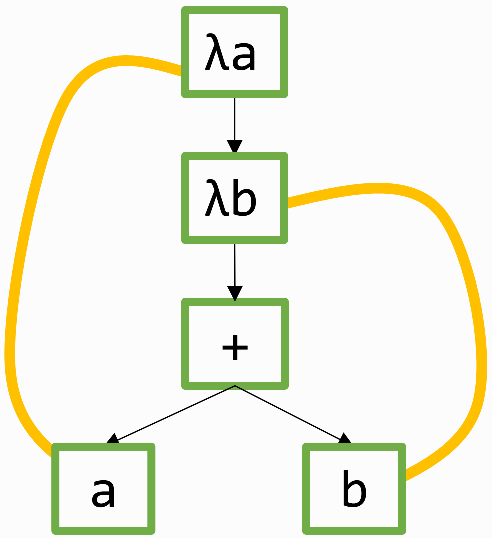
sli.do 1275413

α-equivalence for L_{Lam}

α-equivalence allowed us to equate expressions with let-binders up to renaming of bound variables. We can do exactly the same for lambda expressions.

$$\lambda x.\ \lambda y.\ x + y \quad \approx_\alpha \quad \lambda a.\ \lambda b.\ a + b$$





sli.do 1275413

Semantics: Informal Description

- A function on its own shouldn't reduce further: we shouldn't be reducing the **body** of a function without being given an argument
  - (Non-examinable caveat: there are times in PL theory where allowing reduction under binders is sometimes useful for reasoning – but it is impractical for real PL designs/implementations)
- To evaluate a function application `M N`, we:
  - Evaluate the function down to a lambda expression (`λx. M`)
  - Evaluate the argument down to a value V
  - Replace all occurrences of x in `M` with V, and evaluate the result

sli.do 1275413

Semantics: Informal Description

- A function on its own shouldn't reduce further: we shouldn't be reducing the **body** of a function without being given an argument
  - (Non-examinable caveat: there are times in PL theory where allowing reduction under binders is sometimes useful for reasoning – but it is impractical for real PL designs/implementations)
- To evaluate a function application `M N`, we:
  - Evaluate the function down to a lambda expression (`λx. M`)
  - Evaluate the argument down to a value V
  - Replace all occurrences of `x` in `M` with V, and evaluate the result

This **evaluation strategy** of evaluating the argument before substituting into & running the function body is a **design choice**.

Haskell (for example) instead uses **lazy evaluation** that only evaluates a term **when it is needed**.

sli.do 1275413

Semantics: Informal example

```
(\x. x * x) (10 + 15)
```

Begin by evaluating function (\x . x * x): it's already a value

```
(\x. x * x) 25
```

Evaluate 10 + 15 down to the value 25

```
25 * 25
```

Replace every occurrence of 'x' in the function body with the argument, 25

```
625
```

Evaluate the (now closed) function body to get the final result

sli.do 1275413

L_{Lam} Operational Semantics

First evaluate L to a function $\lambda x. N$

Next, evaluate argument M to a value V

Finally, substitute V for x in the body N, and evaluate it to a value W

$$\frac{\boxed{L \Downarrow \lambda x. N} \quad \boxed{M \Downarrow V} \quad \boxed{N\{V / x\} \Downarrow W}}{\boxed{L\ M} \Downarrow \boxed{W}}$$

To evaluate function L applied to argument M...

The result of the expression is W

- The rule follows the informal description, and is similar in spirit to the rule for evaluating let-bindings
- **However**... There are some caveats with substitution, since functions can contain free variables

sli.do 1275413

We need to be careful with substitution…

Suppose we have some variable `myInt` (in practice, provided as a system environment variable or something similar). Then what happens if we evaluate the following?

```
(λf. λmyInt. f) (λx. x + myInt)
```

$\Downarrow$ Evaluate the application by calculating `(λmyInt. f) {(λx. x + myInt) / f}`

```
λmyInt. (λx. x + myInt)
```

sli.do 1275413

We need to be careful with substitution…

Suppose we have some variable `myInt` (in practice, provided as a system environment variable or something similar). Then what happens if we evaluate the following?

```
(λf. λmyInt. f) (λx. x + myInt)
```

⇓ Evaluate the application by calculating `(λmyInt. f) {(λx. x + myInt) / f}`

```
λmyInt. (λx. x + myInt)
```

Whereas `myInt` was free before, it has now been **captured** by the binder after substitution.

This changes the meaning of the program!

sli.do 1275413

Variable Capture

This is an example of **variable capture**: where a free variable becomes bound after substitution. This can **change the meaning** of the expression. Let's look again...

$$\underbrace{(\lambda f.\ \lambda myInt.\ f)}_{\text{Take a function } f\text{, take another argument which we ignore, then return } f} \quad \underbrace{(\lambda x.\ x + myInt)}_{\text{Take a number } x\text{, then add it to our existing } myInt \text{ number}}$$

$$\underbrace{\lambda myInt.\ (\lambda x.\ x + myInt)}_{\text{Take a new } myInt \text{ number, then return a function that adds a number 'x' to the new } myInt \text{ number}}$$

sli.do 1275413

Avoiding Variable Capture

Variable capture is **never desirable** and we need to find a way to avoid it. Fortunately there is a straightforward solution:

```
(λf. λmyInt. f)   (λx. x + myInt)
```

Whenever we need to substitute under a binder, if we pick fresh names (unused elsewhere) for each of the binders, to generate an α-equivalent expression:

```
(λbob. λroger. bob)   (λx. x + myInt)
```

Since we know `roger` is fresh, we know that it definitely **won't** be free in the argument, and we can substitute without fear of variable capture

```
(λroger. (λx. x + myInt))
```

sli.do 1275413

Capture-Avoiding Substitution (Function cases)

```
(λx. M) { N / x } = λx. M
```

We know there will be **no free occurrences** of x (as all will be bound by the lambda), so can stop.

```
(λx. M) { N / y } = (λx . M {N / x})
```

if $x \neq y$ and $x \notin fv(N)$

We restrict substitution to only be defined if variable capture doesn't occur

sli.do 1275413

Implementing Capture-Avoiding Substitution (1)

- The previous definition is **not defined** when variable capture would occur.
- However, we can always make substitution safe when implementing it by applying the variable freshening trick we saw.
- Let `M(x ↔ y)` be a **swapping** operation that renames x to y, and y to x, in M, for example:
  (**let** x = 5 **in** x + y)(x ↔ y) = **let** y = 5 **in** y + x
  - Full definition in the lab sheet
- We next define `subst(M, N, x)` as the operation to substitute N for x in `M`, freshening variables where required

sli.do 1275413

Implementing Capture-Avoiding Substitution (2)

- The full definition of `subst(M, N, x)` is again in the lab sheet, and most of the cases are straightforward:
  - If M is the variable x, then put N there instead
  - For binary operations and function application, substitute into both subterms
- The interesting cases are for functions:

As before, if we're substituting for the same variable as the binder, we know there will be **no free occurrences** of x, so can stop.

```
subst(λx. M, N, x) = λx. M
```

sli.do 1275413

Implementing Capture-Avoiding Substitution (2)

- The full definition of `subst(M, N, x)` is again in the lab sheet, and most of the cases are straightforward:
  - If M is the variable x, then put N there instead
  - For binary operations and function application, substitute into both subterms
- The interesting cases are for functions:

Generate a fresh variable z and change the binder

Swap all old occurrences of x in the body for the new variable z

```
subst(λx. M, N, y)  = λz. (subst(M(x ↔ z), N, y))
```

(where y ≠ x and z is fresh)

Substitute fearlessly!

sli.do 1275413

Aside: Substitution vs. Environment-Based Interpreters

- Our operational semantics for both L_{Let} and L_{Lam} defined using **substitution**, which is convenient for the theory
- The lab will get you to write a **substitution-based interpreter** following the operational semantics
  - This is **deliberate**: it should help you better understand capture-avoiding substitution, and see the interplay between theory and practice
- In practice, though, substitution-based interpreters are **inefficient** since they require **linear traversals through the AST**
- Instead, most practical interpreters are **environment-based** and keep a mapping from variables to values;
  - Substitution involves extending the environment
  - 'Evaluating' a variable then becomes an environment lookup

The λ-calculus

- Our L_{Lam} calculus is very close to the **untyped λ-calculus** first introduced by Alonzo Church.
- We will discuss recursive functions as a separate construct (as later on we want to consider **typed** languages), but the untyped λ-calculus can express recursion natively!
  - This is due to an intriguing term known as the Y-combinator
- Theory of Computation (hopefully running again next year) goes into **much** more depth on theoretical aspects of the λ-calculus

sli.do 1275413

Question: What do you think happens when you try to run the following?

```
(λx. x x) (λx . x x)
```

sli.do 1275413

Recursion with let-rec

let fun

```
let fun square x =
  x * x
in
square 5 + square 10
```

let rec

```
let rec fac n =
  if n <= 1 then 1 else
  n * (fac (n - 1))
in fac 10
```

- Recall our 'square' example from the start of the lecture
- We can't use **`let fun`** (or lambdas) **recursive** functions: here we can't refer to square in the body of the function definition
- We can instead make a **`let rec`** construct that allows us to refer to the function recursively in its definition.
  - We can therefore write, e.g., factorial, as on the left (with the assumption $n \geq 0$)

sli.do 1275413

Anonymous recursive functions

```
rec f(x). M
```

```
rec fac(n).
  if n <= 1 then 1 else
  n * (fac (n - 1))
```

```
let rec f x = M in N
        ⟺
let f = rec f(x). M in N
```

- As with **`let fun`** and lambda expressions, we can also have **anonymous recursive functions**
- These are just like lambdas, except we give the function a name that is also bound in the function body
- **`let rec`** can also be desugared into anonymous recursive functions
- We refer to the language with **`rec`** functions as L_{Rec}

sli.do 1275413

L_{Lam} Operational Semantics

$$\frac{L \Downarrow \mathbf{rec}\ f(x).M \qquad M \Downarrow V \qquad N\{\mathbf{rec}\ f(x).M / f,\ V / x\} \Downarrow W}{L\ M \Downarrow W}$$

First evaluate L to an anonymous recursive function

Next, evaluate argument M to a value V

Finally, substitute **a copy of the function** for f, and the argument for V, and evaluate to W

To evaluate function L applied to argument M…

The result of the expression is W

- The rule for evaluating an anonymous recursive function application is similar to evaluating a usual function application
- The main difference is that we substitute a **copy** of the **rec** function when evaluating the body

sli.do 1275413

Where we're up to now

- Given (recursive) functions and conditionals, we can now write some more interesting (pure) programs!
  - For example, we can now write programs like Fibonacci, Factorial in our language.
- The big thing we're missing is (recursive) data types – but we won't have time for that in this course, unfortunately
- Next week, we'll look at **types and typechecking**: how can we rule out certain classes of runtime errors without running a program?
- My final week will then talk about **imperative programming**

sli.do 1275413

Overview of this week's lab

- This week's lab is in two parts
- In Part 1, you'll implement a substitution-based interpreter for L_{Rec}
    - Swapping
    - Substitution
    - Interpreter
- In Part 2, you'll implement **desugaring** passes for let-fun, let-rec, and let, which will allow you to specify the behaviour of these constructs *without* explicitly writing interpreter cases for them
- After finishing it, your language should be able to run recursive functions (e.g. Factorial, and you're welcome to write some of your own)

sli.do 1275413

Conclusion

- In this lecture we've seen:
  - How to define functions with let-fun and let-rec
  - Anonymous functions, and anonymous recursive functions
  - Capture-avoiding substitution via binder freshening
- **Next week:** Types, typechecking, and type soundness
- See you this afternoon for the labs!

Answers to sli.do questions

sli.do 1275413

Sli.do Questions (1)

- Why are both Values and Constants defined as n | b ? Are constants different from values?
    - In the cases of the languages we've seen so far, indeed values and constants coincide. I've separated them for a couple of reasons, though.
    - First: it's not always the case that values are a subset of the terms in the language. For example, in an environment-based interpreter, a lambda evaluates to a **closure** that records its current environment. This can't be written by the user.
    - Second: I wanted to make it very clear that you can only get values by evaluating expressions: this makes the big-step evaluation judgement clearer, and also is more closely related to the lab code.

sli.do 1275413

Sli.do Questions (2)

- Will `let x = (let y = 5 in y) in x` evaluate to 5?
  - Yes. Evaluate the inner let first: this will end up as 5. Then, substitute 5 for x, to give the final result of 5.
- Looking at the slides, does this mean our interpreter correctly identify tabs like python?
  - We're working at the level of abstract syntax, so we assume parsing has already been done. Any indentation on the slides is just to make things easier to read.
- Will we be asked to do a reduction example like we did just now in an exam
  - You won't be asked to draw a derivation tree, since this is difficult to do on Moodle (where your exam will be). We would expect you to understand how big-step semantics work, though, maybe explaining informally how an expression would reduce.

sli.do 1275413

Sli.do Questions (3)

- Are 2 expressions α-equivalent if the binding structure the same but the numbers are different?
  - No – for example 5 and 6 have the same binding structure (none ☺) but would not be α-equivalent because they are syntactically different.
- Does swapping the let-binding around also cause two things to not be alpha equivalent ?
  - Indeed – even in places where the order doesn't matter for the result, such as:
    - let x = 1 in let y = 2 in x + y
    - let y = 2 in let x = 1 in x + y
  - These two expressions are **not** α-equivalent
  - I'd encourage you to draw the binding diagrams to see why

sli.do 1275413

Sli.do Questions (4)

- Is the strategy of renaming two terms to try and equate them to a third target term more correct than renaming one of the terms to target the other?
  - This would also be a perfectly good way of doing showing alpha-equivalence (neither approach is more or less correct)
- Is there a case where changing more than the variable names still results in alpha equivalence
  - No – changing anything more than variable names would make the expressions structurally different
- Do the classes of expressiveness that you talked about have anything to do with the classes of computability?
  - Good question, but the two notions are unrelated.

sli.do 1275413

Sli.do Questions (5)

- Could we replace some of the binary operators in L_lam with lambdas? Like multiplication with some recursive lambda using addition?
  - Interesting question! There are a couple of answers to this.
  - You can define multiplication using recursion and addition (if you permit me the encodable use of multi-argument lets -- this assumes y is a positive integer):
  - ```
    rec mult(x, y) =
      if y == 1 then x else x + (mult x (y - 1))
    ```
  - You can actually do addition and multiplication **just** using lambdas and applications using an approach known as Church Encoding
    ```

sli.do 1275413

Sli.do Questions (6)

- Why can you give a lambda function without an argument, whereas you can't give a let without an 'in'?
  - The main thing with a let is that it doesn't make sense without an 'in'.
  - Suppose we just had "let x = M" – this doesn't have a scope, and we don't know what to evaluate next afterwards
  - On the other hand, (\x. M) is a function, and we can apply this separately (e.g. (\x. M) 5)
- Lazy evaluation allows you to do cool stuff like infinite lists
  - Indeed it does! There are tons of interesting consequences (and tradeoffs) of lazy evaluation / call-by-name / call-by-need. Alas there's not enough time for me to explore these in depth this time ☹

sli.do 1275413

Sli.do Questions (7)

- In the informal example, we dont need to evaluate the function as it is already a value. When would it not be a value?
  - Good question – consider the following expression:
  - ((\x . x) (\y . y + 5)) 10
  - The function here isn't a value yet – we need to evaluate the application down to get (\y . y + 5) 10
  - Now it's a value, so we can evaluate it down to 15
- Is there a case for variable capture to be intended?
  - I'm not aware of any cases where variable capture is desirable.
- Do we refer to let rec as recursion bindings?
  - I tend to call let-rec a "recursive let binding", and `rec f(x).M` an "anonymous recursive function".