CSCI 334:
Principles of Programming Languages

Lecture 4: Fundamentals II

Instructor: Dan Barowy

Williams

Activity

Write a function `firsts` that, when given a list of `cons` cells, returns a list of the left element of each `cons`.

```
( (a . b) (c . d) (e . f) (g . h) )
                 ↓
              firsts
                 ↓
             (a c e g)
```

What is computable?

- Hilbert: Is there an algorithm that can decide whether a logical statement is valid given axioms?
- “Entscheidungsproblem” (literally “decision problem”)
- Leibniz thought so!

What is computable?

- Why do we care?
- $f(x) = x + 1$
- We can clearly do this with pencil and paper.
- $\int 6x\,dx$
- Also computable, in a different manner.
- We care because the computable functions can be done on a computer.

"What is the answer to the ultimate question of life, the universe, and everything?

Lambda calculus

- Invented by Alonzo Church in order to solve the Entscheidungsproblem.
- Short answer to Hilbert's question: no.
- Proof: No algorithm can decide equivalence of two arbitrary **λ**-calculus expressions.

Lambda calculus is deceptively simple

- Church-Turing thesis: every computable function can be represented in the **λ**-calculus; i.e., it is "Turing complete".
- Grammar in BNF:

```
M ::= x          variable
    | λx.M       abstraction
    | MM         function application
```

```
M ::= x     →    variable
    | λx.M       abstraction
    | MM         function application
```

- "pure" **λ**-calculus doesn't say anything about the *values* of variables.
- We often extend **λ**-calculus with arithmetic, so "`1`", "`2`", "`3`", … are also considered terms.
- Justified by Church's own proof that arithmetic (Peano axioms) can be encoded in **λ**-calculus (see "Church encoding").

```
M ::= x          variable
    | λx.M  ⟶   abstraction
    | MM         function application
```

- Functions are at the heart of **λ**-calculus.
- Functions are "nameless":
  - Just a λ denoting "function".
  - A "bound variable" `x ∈ {x, y, z, …}`
  - An expression `M` where `x` is "bound".
- E.g., `λx.x + 1` adds one to `x`

```
M ::= x          variable
    | λx.M  ⟶   abstraction
    | MM         function application
```

- `λx.x + 1`
- Translation:

  def func(x):

  return x + 1
- Remember that no programming languages existed at the time. **λ**-calculus was the first!
- Why "**λ**"?

```
M ::= x          variable
    | λx.M       abstraction
    | MM    ⟶   function application
```

- How do we compute 5 + 1 = 6?
- `(λx.x + 1)5`
- Works by process of substitution

```
  ([5/x]x + 1)
= (5 + 1)
= 6
```

α-equivalence

- The chosen symbol for a bound variable does not matter.
- $\lambda x.x =_\alpha \lambda y.y$
- More precisely

  `λx.M = λy.[y/x]M`
- "Substitute `y` for occurrences of `x` in `M` (such that y does not already appear in `M`)."
- `M` is the "scope" of the binding.

Free variables

- These two are not **α**-equivalent
- $\lambda x.x + b \neq_{\alpha} \lambda y.y + c$
- Why? b and c are "free variables"
- Proof:

  $\lambda x.x + b$

  $=_{\alpha} ([y/x]\lambda x.x + b)$

  $= \lambda y.y + b$
- $\lambda y.y + b \neq_{\alpha} \lambda y.y + c$

β-equivalence

- We compute function application using substitution: "**β**-reduction"
- $(\lambda x.x + 1)5 =_{\beta} 5 + 1$
- How did we get that?
- We substituted 5 for x.
- $(\lambda x.x + 1)5$

  $=_{\beta} ([5/x]x + 1)$

  $= (5 + 1)$

  $= 5 + 1$

Constant function

- A constant function is one that does not depend on a variable
- $(\lambda x.1) = 1$
- Means that unnecessary variables can be eliminated
- $(\lambda x.\lambda y.y) = (\lambda y.y)$

η-equivalence

- If an abstraction exists solely to pass its argument to another function, the abstraction can be eliminated.
- $\lambda x.M\ x =_{\eta} M$

  (assuming that x does not appear in M)

Renaming bound variables

- Some expressions are hard to evaluate unless you rename some of the variables.
- `(λf.λx.f(f x))(λy.y + x)`
- Note that the free variable x appears on the right and the bound variable x appears on the left. *These are different variables!*
- `([z/x]λf.λx.f(f x))(λy.y + x)`

  $=_\alpha$ `(λf.λz.f(f z))(λy.y + x)`

  … `(λz.z + x + x)`

Order does not matter

$$\begin{tikzcd}
& M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\
& N &
\end{tikzcd}$$

If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

“confluence”

Activity

`(λf.λx.f(f x))(λz.x+z)2`

…

`= x + x + 2`