Programming languages

Important languages

Fortran – First lang w/ a compiler, syntax is whatever the compiler accepts, no formal standards set. **Lisp** – Computing abstract / symbolic stuff, dynamic scope for variables. **COBOL** – Used for "business" programming, verbose and made to read like English, good layout capabilities. **Algol 60** – First language with a standard (BNF invented for this), designed by committee. **ML** – Abstract data types, hides details about types from programmer, can't lie to compiler, tradeoff of more debugging for less hassle down the line.

Esoteric languages

Esolangs are programming languages designed to experiment with weird ideas, made intentionally hard to program in, or as a joke, rather than practical use. Esolangs are known for breaking modern coding conventions to possible innovate with new ideas. Some examples of esolangs are Shakespeare, Befunge (2D text syntax), LOLCODE, Malbolge (designed to be unwritable - 'crazy operation', base-three arithmetic, and self-altering code), INTERCAL (satire of programming languages and new notation), Brainfuck (minimal, unreadable turing machine simulator), and PIET (syntax as blocks of colour in an image).

Syntax

How a programming language "looks". Often a string, but can be a picture (Monet), or a grid of cells (Excel).

BNF

Formal specification of string-based syntaxes.

$\langle e \rangle$ ::= x
| $\lambda x \langle e \rangle$
| $\langle e \rangle\ \langle e \rangle$
| $(\langle e \rangle)$

Dynamic Semantics

Substitution

$[x \mapsto s]x = s \qquad x[v/x] = v$

Evaluation Strategies

Can't evaluate anything under a lambda in both of the below strategies.

Full Beta-Reduction:

beta-reduce in any order: non-deterministic.

Normal Order:

reduce the leftmost, outermost expression until no more expressions.

Call by Name

– Evaluate function calls without evaluating arguments. Stop when the outermost term is a lambda.

$$\frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2} \qquad \frac{}{(\lambda x.e_1)e_2 \to e_1[e_2/x]} \qquad \frac{}{\text{fst}(e_2, e_2) \to e_2}$$

$$\frac{}{\text{snd}(e_1, e_2) \to e_2} \qquad \frac{e_1 \to e_1'}{\text{and } e_1 e_2 \to \text{and } e_1' e_2} \qquad \frac{}{\text{and true } e_2 \to e_2}$$

$$\frac{}{\text{and false } e_2 \to \text{false}}$$

$$\frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3}$$

$$\frac{}{\text{if true then } e_2 \text{ else } e_3 \to e_2} \qquad \frac{}{\text{if false then } e_2 \text{ else } e_3 \to e_3}$$

$$\frac{}{\text{let } x = e_1 \text{ in } e_2 \to e_2[e_1/x]}$$

Call by Value

– Evaluate arguments to function calls before evaluating the function call itself. Stop when the outermost term is a lambda. This is the evaluation strategy most languages adopt.

$$\frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2} \qquad \frac{e_2 \to e_2'}{v_1 e_2 \to v_1 e_2'} \qquad \frac{}{(\lambda x.e_1)v_2 \to e_1[v_2/x]}$$

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')} \qquad \frac{}{\text{fst}(v_1, v_2) \to v_1}$$

$$\frac{}{\text{snd}(v_1, v_2) \to v_2} \qquad \frac{e_1 \to e_1'}{\text{and } e_1 e_2 \to \text{and } e_1' e_2} \qquad \frac{}{\text{and true } e_2 \to e_2}$$

$$\frac{}{\text{and false } e_2 \to \text{false}}$$

$$\frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3}$$

$$\frac{}{\text{if true then } e_2 \text{ else } e_3 \to e_2} \qquad \frac{}{\text{if false then } e_2 \text{ else } e_3 \to e_3}$$

$$\frac{e_1 \to e_1'}{\text{let } x = e_1 \text{ in } e_2 \to \text{let } x = e_1' \text{ in } e_2}$$

$$\frac{}{\text{let } x = v_1 \text{ in } e_2 \to e_2[v_1/x]}$$

Church Encodings

Booleans

tru = $\lambda t.\lambda f.t$
fls = $\lambda t.\lambda f.f$
and = $\lambda b.\lambda c.bc$ fls
or = $\lambda b.\lambda c.b$ tru c not = $\lambda p.p$ fls tru

Numerals

$0 = \lambda s.\lambda z.z$
$1 = \lambda s.\lambda z.sz$
$2 = \lambda s.\lambda z.s(sz)$ plus = $\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$
succ = $\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
mult = $\lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)\ x$
exp = $\lambda m.\lambda n.n\ m$
pred = $\lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u)$
pred = $\lambda m.fst(msszz)$ where
$ss = \lambda p.pair(sndp)(plusc_1(sndp))$ and $zz = pairc_0c_0$
minus = $\lambda m.\lambda n.(n$ pred$)\ m$
iszero = (minus $n\ m$)
iszero = $\lambda m.m(\lambda x.$ fls)tru

Pairs

pair = $\lambda x.\lambda y.\lambda z.z\ x\ y$
fst = $\lambda p.p(\lambda x.\lambda y.x)$
snd = $\lambda p.p(\lambda x.\lambda y.y)$

Either

left = $\lambda a.\lambda l.\lambda r.la$
right = $\lambda b.\lambda l.\lambda r.rb$

Lists

nil = $\lambda c.\lambda n.n$
cons = $\lambda h.\lambda t.\lambda c.\lambda n.c\ h\ (t\ c\ n)$
isnil = $\lambda l.l(\lambda h.\lambda t.$ fls) tru
head = $\lambda l.l(\lambda h.\lambda t.h)$ fls
tail = $\lambda l.\lambda c.\lambda n.l(\lambda h.\lambda t.\lambda g.gh(tc))(\lambda t.n)(\lambda h.\lambda t.t)$

Trees

leaf = $\lambda x.\lambda b.\lambda l.l, x$
branch = $\lambda x.\lambda y.\lambda b.\lambda l., b, (x, l, b), (y, l, b)$

Maybe

nothing = $\lambda n.\lambda j.n$
just = $\lambda a.\lambda n.\lambda j.ja$
isNothing = $\lambda n.n(\lambda_.$true$)(\lambda_.$false$)$
isJust = $\lambda n.n(\lambda_.$false$)(\lambda_.$true$)$
maybe = λdef.$\lambda f.\lambda m.m$def$(\lambda a.\ f\ a)$

Domain Specific Languages (DSLs)

Shallow

Implement the DSL as functions. Ex. for pic DSL, stuff is given as functions. Easy to add new functions, but can suck if you want to extend to more interpretations since you can't reuse functions. Pros:
- easy extensibility of terms
- Has nice syntax due to being programmed in an established language
- piggyback on host language's type system (pro in some cases)

Cons:
- no reuse of ASTs
- Hard to re-interpret terms with other meanings (e.g. hard to change format of a picture)
- things are immediately evaluated
- piggyback on host language's type system (con in some cases), e.g. if you want to move your dsl to another language
- Cannot easily perform program transformations (e.g. optimizations)

Deep

Datatype to represent syntax. Can easily write new interpretation functions. But annoying to add new operations because all interpretations need to be extended. Pros:
- easy reuse of programs written in the DSL
- custom validity system
- simulatable
- domain-specific interpretations, e.g. tool creation, analysis, re-writing, optimization passes, etc.
- Can easily perform program transformations (e.g. optimizations)

Cons:
- Ugly syntax
- extra tag i.e. the DSL is a little less efficient
- term set is typically not 'open'
  - we can't easily extend a language without recompilation and defining interpreation of new terms for all previously defined operators
  - expression problem (? related to above?)

Tagless

Best of shallow and deep. The language is the interpretation. If we want to add a new operation we can easily add a new class that extends the original. To create a new interpretation just create an instance of the class. Pros:
- Both of the pros from shallow and deep embeds
- Programming against an abstract interface is nice

Cons:
- Speed: the compiler does not know what you are working against with an abstract interface so compiler optimizations might not work properly

Static Semantics (Typing)

Type safety is an assurance that computations do not lead to mismatched types in other computations — program execution does not lead to ill-defined states. A **type system** consists of two pieces; judgements and inference rules. A judgement is some property that the type system lets us show; for instance, we might have a judgement that asserts that some term x has type A. The inference rules allow us to actually prove that some judgement holds. For instance, this is how one might prove that (true, false) has type Bool × Bool. DSLs are particularly neat because they allow for domain-specific: analysis, validation, interpretation, optimization, tooling, etc.

Progress

Definition: a well-typed term is either a value or, or may be evaluated once under single-step operational semantics. That is, a well-typed term is not stuck.

Preservation

Definition: if a term is well-typed, and we evaluate it once under single-step operational semantics, the resulting term is also well typed.

Context

Unification

Unification Algorithm

unify(C) =
if $C = \emptyset$, then [ ]
else let $\{S = T\} \bigcup C' = C$ in
if $S = T$
  then unify(C')
else if $S = X$ and $X \notin FV(T)$ (If S is unbounded in T:)
  then unify($[X \mapsto S]C' \circ [X \mapsto T]$
else if $T = X$ and $X \notin FV(S)$ (If T is unbounded in S:)
  then unify($[X \mapsto T]C') \circ [X \mapsto S]$
else if $S = S_1 \to S_2$ and $T = T_1 \to T_2$
  then unify($C' \cup \{S_1 = T_1, S_2 = T_2\}$)
else
  fail

Unification Less Yap

if $C = \emptyset \implies$ []
  Attempt to get $S = T$ from C :
    $C' = C - \{S = T\}$
    If $S = T$ directly:
      $\implies$ unify(C')
    If S is unbounded in T:
      $\implies$ unify($C'[S \mapsto T]$) $\circ\ [S \mapsto T]$
    If T is unbounded in S:
      $\implies$ unify($C'[T \mapsto S]$) $\circ\ [T \mapsto S]$
    If S=T follows $s_1 \to s_2 = t_1 \to t_2$:
      $\implies$ unify($C' + \{s_1 = t_1, s_2 = t_2\}$)
  If no $S = T \subseteq C$:
    Choose $S = s_1 \to t_1$ and $T = t_1 \to t_2$:
    $C' = C - \{S, T\}$
    $\implies$ unify($C' + \{s_1 = t_1, s_2 = t_2\}$)

Subtyping

$S <: T$ says 'S is a subtype of T'.
Unlike 'subsets' a subtype is 'more informative', meaning it may contain more parameters, but never fewer. So a subtype can always be used safely in place of the 'supertype.'
e.g. $S = \{x : \text{Nat}, y : \text{Nat}\}$, $T = \{x : \text{Nat}\}$, so $S <: T$.

Rules

T-Sub: $\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$
'subtype can be typed as its supertype, too'
S-Refl: $S <: S$
'everything is a subtype of itself'
S-Trans: $\frac{S <: U \quad U <: T}{S <: T}$
'subtype relation is transitive'
S-RcdWidth: $\{l_i : T_i^{i \in 1..n+k}\} <: \{l_i : T_i^{i \in 1..n}\}$
'we can add extra fields in a record, still a subtype'
S-RcdDepth: $\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i^{i \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}}$
'can also add depth-wise (within each field's fields)'
S-RcdPerm: $\frac{\{k_j : S_j^{j \in 1..n}\} \text{ is perm. of } \{l_i : T_i^{i \in 1..n}\}}{\{k_j : S_j^{j \in 1..n}\} <: \{l_i : T_I^{i \in 1..n}\}}$
'order of fields does not matter (can be a permutation)'
S-Arrow: $\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$
'a function that takes in less information and outputs more, is a subtype of a function that takes in more information and outputs less'
Definition: $S <: T$ means that an element of S may be safely used whereever an element of T is expected.
This S-Arrow subtype relation is **contravariant** in the left-hand sides of arrows , and **covariant** in the right-hand sides of arrows. The intuition is that if we have a function $S_1 \to S_2$, then it will accept any subtype of S_1 as input, and the result S_2 can be viewed as belonging to any supertype of S_2.
S-Top: $S <:$ Top, where Top is a new type constant that is a supertype of every type.

Definition: Liskov's substitution principle: If X inherits from Y, then X should pass all of Y's black box texts. Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. A function works on all subclasses of a class.
Rules: (1) Inherited methods of a class must not strengthen preconditions or weaken postconditions; it must accept a superset of the values the parent method accepts and output a subset of the parent's possible outputs. The parameters must be contravariant and the return values covariant. (2) Subtypes must not weaken the class invariants. (3) History rule: the subtype cannot change in a way prohibited by the supertype,

Q&A

Q1 Practice Answer

CBN

$((\lambda x.\lambda y.yy)((\lambda w.ww)(\lambda w.ww)))(\lambda w.w)$
//λx is called, all x's replaced
$(\lambda y.yy)(\lambda w.w)$
//λy is called, all y's replaced w/ $(\lambda w.w)$
$(\lambda w.w)(\lambda w.w)$
//λw is caled, all w's replaced w/ $(\lambda w.w)$
$(\lambda w.w)$

CBV

$(\lambda x.\lambda y.yy)((\lambda w.ww)(\lambda w.ww))(\lambda w.w)$
$(\lambda x.\lambda y.yy)((\lambda w.ww)(\lambda w.ww))(\lambda w.w)$
//infinite loop

Q2 - CBV rules given CBN rules for pairs and projections

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')}$$

$$\frac{e_1 \to e_1'}{\text{fst } e_1 \to \text{fst } e_1'} \qquad \frac{e_1 \to e_1'}{\text{snd } e_1 \to \text{snd } e_1'}$$

$$\frac{}{\text{fst } (v_1, v_2) \to v_1} \qquad \frac{}{\text{snd } (v_1, v_2) \to v_2}$$

CBV requires the full evaluation of the expression under fst/snd before operating on it. This means that the pair fst/snd is operating over must contain two expressions that can be fully evaluated.

Q3 - Fix Incoherent Rules

$$\frac{e_1 \to e_1'}{\text{and } e_1\ e_2 \to \text{and } e_1'\ e_2} \qquad \frac{}{\text{and true } e_2 \to e_2}$$

$$\frac{}{\text{and false } e_2 \to \text{false}}$$

The first rule is for and checks if e_1 can be evaluated. The old rules checked if the second term was true or false, meaning that e_2 would have to be the term true or false since there is no rule to evaluate e_2 to see if it is true or false.

Q8 - Expression evaluates in more steps under cbv

$(\lambda x, y.xxy)((\lambda x.x)b)((\lambda x.x)b)$
under cbn:
$(\lambda x, y.xxy)((\lambda x.x)b)((\lambda x.x)b)$
$= (\lambda y.((\lambda x.x)b)((\lambda x.x)b)y)((\lambda x.x)b)$
$= ((\lambda x.x)b)((\lambda x.x)b)((\lambda x.x)b)$
$= b((\lambda x.x)b)((\lambda x.x)b)$
under cbv:
$(\lambda x, y.xxy)((\lambda x.x)b)((\lambda x.x)b)$
$= (\lambda x, y.xxy)b((\lambda x.x)b)$
$= (\lambda x, y.xxy)bb$
$= (\lambda y.bby)b$
$= bbb$
alt answer
$(\lambda x.a)((\lambda y.y)b)$
CBV (2 steps):
$(\lambda x.a)((\lambda y.y)b)$
$(\lambda x.a)(b) \to a$
CBN (1 step): $(\lambda x.a)((\lambda y.y)b) \to a$ | $(\lambda x.\lambda y.\ y)$(anything very long that can evaluate)
In CbV this will need to evaluate the very long term, but in CbN it will try to substitute it exactly for x, but x is not used, so the evaluation does not need to be done.

Q9 - Expression evaluates in more steps under cbn

alt answer
Justify that your example is correct. $(\lambda x.xx)((\lambda x.x)(\lambda y.y))$
CbV: $(\lambda x.xx)((\lambda x.x)(\lambda y.y)) \rightarrow (\lambda x.xx)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y.y)$
CbN: $(\lambda x.xx)((\lambda x.x)(\lambda y.y)) \rightarrow ((\lambda x.x)(\lambda y.y))((\lambda x.x)(\lambda y.y)) \rightarrow (\lambda y.y)((\lambda x.x)(\lambda y.y)) \rightarrow (\lambda x.x)(\lambda y.y) \rightarrow (\lambda y.y)$ (or) $(\lambda y.\ yyyyy)$(any lambda expression that can evaluate)
CbV will evaluate the second term once and then substitute the value of it, but CbN will substitute it exactly into every y. B/c of this, the evaluation will need to be done 5 times.

Q10 - Expression evaluates in exactly 2 steps under both but then gets stuck

Note 1: stuck means that a term evaluates to a normal form (no other possible eval rules can be applied) but not a value (as defined by bnf usually).
Note 2: This solution can be extended for any number of eval steps greater than 2. Add as many identity functions in between the first two terms as you want to increase the number of eval steps by the same amount.

$$(\lambda x.x)((\lambda x.x)z)(\lambda x.x)$$

under cbn:

$$= ((\lambda x.x)z)(\lambda x.x)$$

$$= z(\lambda x.x)$$

under cbv:

$$= (\lambda x.x)z(\lambda x.x)$$

$$= z(\lambda x.x)$$

alt answer
When a term is in normal form, but not a value, it is stuck.
Example Expr: and True $((\lambda x.x)a)$
CBN: 1 : $\lambda x.xa \rightarrow$ 2 : a
CBV: 1. and True a $\rightarrow$ 2. a

Question 11

Explain why the Ω combinator $(\lambda x.x\ x)(\lambda x.x\ x)$, using CbN, does not lead to a terminating evaluation.

Under Call-by-Name (CbN), the function $\lambda x.xx$ simply substitutes the argument $(\lambda x.xx)$ wherever x appears in the body of the function without evaluating it. So, the result is $(\lambda x.xx)(\lambda x.xx)$ after one reduction step. As you can see, the result is the same as the original expression. If you try to reduce it further, you'll get the same expression again. This process will continue indefinitely, leading to a non-terminating evaluation

Q13 - What's the point of an eval strategy?

The point of having an evaluation strategy is for you or a computer to know how to evaluate an expression if multiple reductions are possible. We need them for building operational semantics. If there isnt an eval strategy then evaluating an expression would be nondeterministic.

17 - Convention used with respect to bound variables

The convention used is alpha-equivalence. It captures the idea that it's safe to rename a variable in a program if you also fix all the references to that variable. That is, when you change the parameter of a lambda term, you also have to go into the lambda's body and change the usages of that variable. Terms that differ only in the names of bound variables are interchangeable in all context. You can change the name of a bound variable in a statement and the statements before and after the change are the same. e.g. $(\lambda x.x)$ is alpha equivalent to $(\lambda y.y)$.

When writing lambda terms for humans, we use a convention called **alpha-conversion** to avoid confusion between free and bound variables. Simply, this is the process of renaming parameters in functions without changing the meaning of the function. For example, $(\lambda x.x)(\lambda y.yx)$. Here, the first instance of x is bound while the second instance is free. To avoid confusion, we rename the bound variable to get $(\lambda z.z)(\lambda y.yx)$. This renaming method does not change the meaning of the term and makes it easier to read and understand.

In linear algebra, it's common to use alpha conversion renaming when working with subsets and quantifiers of certain sets which contain vairables U V W, where you can modify and manipulate a set and keep its original contents but when working with its modified version to show its still different we rename it in terms of bound variables like (a + b = 0) for all b and there exists an a, is the same as (x + y = 0) for all x and there exists an y. In predicate calculus, we usually do renaming with quantifiers ∀ and ∃ since it sometimes may contain a variable that's not free. In integrals, when you do double integrals, you don't integrate based on the same variable. As long as the mathematical equation is preserved in its meaning, so theres no direct change when substituting values from their bound counter part.

Question 18 (5) - Unification

Unify $a \rightarrow a$, $(b \rightarrow c) \rightarrow (d \rightarrow e)$, and $(d \rightarrow c) \rightarrow a$.

$\{a \rightarrow a = (b \rightarrow c) \rightarrow (d \rightarrow e),\ a \rightarrow a = (d \rightarrow c) \rightarrow a\}|[]$

$\{a = (b \rightarrow c),\ a = (d \rightarrow e),\ a = (d \rightarrow c),\ a = a\}|[]$

$\{a = (d \rightarrow e),\ a = (d \rightarrow c)\}|\sigma_1[a/(b \rightarrow c)]$

$\{(b \rightarrow c) = (d \rightarrow e),\ (b \rightarrow c) = (d \rightarrow c)\}|\sigma_2[] \circ \sigma_1$

$\{b = d,\ c = e,\ b = d,\ c = c\}|\sigma_2[] \circ \sigma_1$

$\{b = d,\ c = e\}|\sigma_2[] \circ \sigma_1$

$\{c = e\}|\sigma_2[b/d] \circ \sigma_1$

$\{c = e\}|\sigma_3[] \circ \sigma_2 \circ \sigma_1$

$\{\}|\sigma_3[c/e] \circ \sigma_2 \circ \sigma_1$

$\sigma = [c/e] \circ [b/d] \circ [a/(b \rightarrow c)]$

Question 18 Full

Answers Vary.

1. unify $\{a \rightarrow I, B \rightarrow b\}$.
   A: [] ∘ $[b \rightarrow B]$ ∘ $[a \rightarrow I]$
2. unify $\{a\ toa, B \rightarrow b\}$.
   A: [] ∘ $[b \rightarrow B]$ ∘ $[a \rightarrow b]$
3. Why can't $\{a, a \rightarrow a\}$ reduce?
   A: because there is no case for the algorithm to continue. Cannot ensure $a \rightarrow a$ holds in regards to a.
4. unify $\{a \rightarrow (a \rightarrow b), (c \rightarrow d) \rightarrow (b \rightarrow b)\}$.
   A: [] ∘ $[b \rightarrow (c \rightarrow d)]$ ∘ $[a \rightarrow (c \rightarrow d)]$
5. unify $\{a \rightarrow a, (b \rightarrow c) \rightarrow (d \rightarrow e), (d \rightarrow c) \rightarrow a\}$.
   A: [] ∘ $[b \rightarrow d]$ ∘ $[c \rightarrow e]$ ∘ $[a \rightarrow (b \rightarrow c)]$

alt explanation
i) $a \rightarrow$ intT & boolT $\rightarrow b$ We use reduction rule #2 {a = boolT, intT = b} so [Int/b] ∘ [Bool/a]
ii) $a \rightarrow a$ & boolT $\rightarrow b$ We use reduction rule #2 a = boolT, a = b so [boolT/b] ∘ [boolT/a]
iii) a & $(a \rightarrow a)$ This is reduction rule #3, where 'x is a and t is $(a \rightarrow a)$. However, this is the case where 'x appears in t, because a appears in $(a \rightarrow a)$. Therefore, it cannot be reduced. You can also think about how if $a = (a \rightarrow a)$, then it also equals $[(a \rightarrow a) \rightarrow (a \rightarrow a)]$, etc. (infinitely recurses)
iv) unify $a \rightarrow (a \rightarrow b)$ and $(c \rightarrow d) \rightarrow (b \rightarrow b)$.
$a = (c \rightarrow d)$ & $(a \rightarrow b) = (b \rightarrow b)$ (based on rule 2)
$\{(c \rightarrow d)/a\}$ & $(a = b) = (b = b))$ (based on rule 3)
$b = (c \rightarrow d)$ (based on substitution) $[(c \rightarrow d)/b] \circ [(c \rightarrow d)/a]$
v) unify $a \rightarrow a$ and $(b \rightarrow c) \rightarrow (d \rightarrow e)$ and $(d \rightarrow c) \rightarrow a$.
$\{(d \rightarrow c)/a\}$ (use reduction rule 3)
$((d \rightarrow c) \rightarrow (d \rightarrow c))$ & $((b \rightarrow c) \rightarrow (d \rightarrow e))$ (by normal substitution rule) $((d \rightarrow c) = (b \rightarrow c))$ & $((d \rightarrow c) = (d \rightarrow e))$ (by rule 2) $(d = b)$ & $(c = e)$ (by equalities)
$[d/b] \circ [c/e] \circ [(d \rightarrow c)/a]$

Question 19

1. $h :: (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b$

   $h\ f\ x = f\ x\ x$
2. $c :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

   $c\ f\ g = \backslash x \rightarrow g\ (f\ x)$
3. $d :: (a \rightarrow b) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow c)$

   $d\ f\ g = \backslash x \rightarrow g\ x\ (f\ x)$
4. $l\ f =$ let $g\ h\ x = h\ (h\ x)$ in $g\ f$

   $x :: a$

   $h :: a \rightarrow a$

   $g :: (a \rightarrow a) \rightarrow a \rightarrow a$

   $l :: (a \rightarrow a) \rightarrow a \rightarrow a$

alt explanation

```
i)  (a -> a -> b) -> a -> b
ii)  (a -> b) -> ( b -> c) -> c
iii) (a -> b) -> (a ->  b -> c) -> c
iv)  (a -> a) -> a
v)  (a -> b) -> b
```

```
i) (a -> a -> b) -> a -> b
(a->a ->b) is type of f
And x is type a
Because we are calling f as output, we return b
```

```
ii)  (a -> b) -> ( b -> c) -> (a -> c)
(a->b) is type for f
G take f output as its input, so its type is (b->c)
Since output is lambda function and its output is calling
       ↪ g function, its output is (a->c)
```

```
iii)  (a -> b) -> (a ->  b -> c) -> (a -> c)
(a->b) is type for f
G take f output as its input, so its type is (a->b->c)
Since output is lambda function and its output is calling
       ↪ g function, its output is (a->c)
```

```
iv) (a -> a) -> (a -> a)
F is type (a->a) since it is same as type for h-function
Since we need input for x, we can write output signature
       ↪ as if it was for a lambda function
v) Not possible
We have two inputs/constants and just applied with each
↪ other,which is impossible
```

Question Q21

Subtyping of function types illustrates both covariance and contravariance - explain those terms.
Subtyping: hierarchical relation on types, <: can be read as ≤ "is less than"
Covariance: The ordering of component types is preserved.
Contravariance: The ordering of component types is reversed.

$$\frac{T_1 <: S_2 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Here S_1, T_1 is contravariance (swapped sides of <:) and S_2, T_2 is covariance (ordering of <: is the same)
Covariance: In a covariant relationship, the subtyping relationship preserves the direction of the original types. If A is subtype of B, then assume we have additional type C, then A → C is a subtype of B → C **Contravariance:** In a contravariant relationship, the subtyping relationship is reversed in direction. If A is subtype of B, then assume we have additional type C, then B → C is a subtype of A → C.

Question 23

Why are the semantics of case-of (for sums) in a CbV language weird? Why is this inevitable? Why are the semantics fine in CbN? Contrast this with 'either' in Haskell.
In CbV we would expect to evaluate the inside fully before the outside. However, this is again not the case, as we first evaluate the case statement, before deciding where to substitute it. This is inevitable, as for sum types, the valid evaluation steps changes based on the specific type we are dealing with.
In CbN we do not evaluate the argument before passing it to begin with, so it is as one would expect.
This is also exactly how Haskell approaches this, as it uses a variant of CbN, and thus lazily evaluates in the case of Either.

Question 24

Subtyping of function types involves contravariance. Explain what that is and give an example.
Let's say we have a number type, an integer type, and a short integer type, where short integer is a subtype of integer and integer is a subtype of number. Then, if we have a function 'integer -> integer', we can use the S-Arrow rule to see that: 1. For the input of the function, we could provide a short integer. This is intuitively valid because if we need to supply an integer, a short integer works. 2. The output could be a number or we could treat it like a number. This is intuitively valid because if the output is an integer, then anything that applies to numbers would also apply to the output. So we can see that 'short integer -> number' is a subset of 'integer -> integer'. In this example, contravariance refers to the left-hand side of the arrow, i.e., the input, where we could use a short integer in the place of an integer.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Note the order of T_1 and S_1 in the first premise. The subtype relation is contravariant in the left-hand side of the arrows and covariant in the right-hand side of the arrows.

Question 25

(Bonus) Given an example of the failure of preservation.
Define everything you need to illustrate this.
- undecidable type checking: type checking in lambda calculus can become undecidable. for example systems that involve recursive types might lead to undecidable.
- polymorphism: systems with polymorphic types, maintaining preservation could be challenging and contradictory.

Miscellaneous

Ω-combinator: $(\lambda x.x\ x)(\lambda x.x\ x)$
Simplest function that recurses infinitely without calling itself.
Y-combinator: $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$
This function generalizes recursion to any other function in lambda calculus.