CS 242 Lecture Notes

Lecture 1. Course overview

Takeaways:

1. appreciation for programming languages as a technical field
2. Becoming a better programmer
3. understand the future
4. survey of some important modern languages
5. basics of active research areas

Language goals have tradeoffs between

- performance
- productivity
- safety

History

- Turing machines and lambda calc to study questions in abstract math and proofs
- Turing:
  - first languages - high level languages translated to machine code
  - Fortran is first real language
  - C language to code operating system
  - Java to improve issues with C++
  - python as language for productivity ontop of C++
- church languages: Lambda -> Lisp -> ML -> Haskell

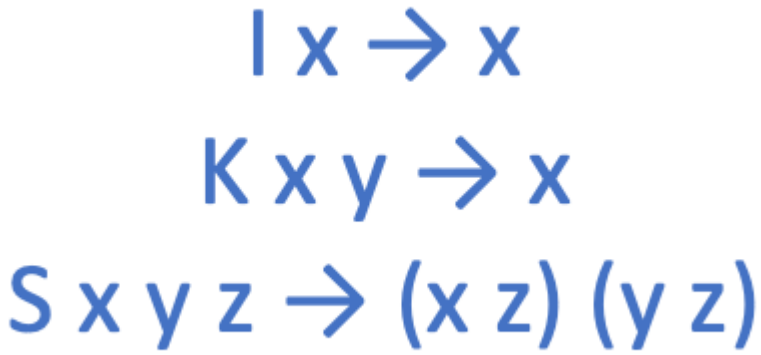
Lecture 2. Combinator Calculus

SKI Calculus

- variable free PL with only functions
- turing complete
- starting point for more involved languages

SKI:

- I : identity
- K: constant function
- S: application / copy function
- terms are trees, not strings. greedy parentheses



K takes first of following two args
SK takes second of following two args
SII duplicates following arg

Recursion

(S I I) (S I I) is a non-terminating expression

- always can be rewritten, since it rewrites eventually to itself

Can do recursive function calls by shoving a term in between the two S I I.
x = S (K f) (S I I)
S I I x -> *x x* -> f (x x) and so on.

Conditionals

true is T, encoded as K
False is F, encoded as S K
not, or, and, if-else can all be encoded.

Abstraction

Use rules to abstract function into SKI

- A(x,x) = I
- A(E,x) = K E if x does not appear in E
- A(E1 E2, x) = S A(E1, x) A (E2, x)

Higher order function: functions can take functions as arguments and return functions as results

- Define
  - c1 x y z = x (y z) – a version of S where the first argument is constant (doesn't use z)
  - c2 x y z = (x z) y – a version of S where the second argument is constant (doesn't use z)

- Add new cases for to the abstraction algorithm for applications that use c1 or c2 if possible

  A(E1 E2,x) = c1 E1 A(E2,x) if x does not appear in E1
  A(E1 E2,x) = c2 A(E1,x) E2 if x does not appear in E2
  A(E1 E2,x) = S A(E1,x) A(E2,x) otherwise

Numbers and arithmetic

n f x is applying f n times to x
succ x = x + 1
add x y = x succ y
mul x y = x (add y) 0
primtive recursion and looping

Lecture 3. Combinators II

Order of evaluation? Reduction strategy.

Most languages have fixed reduction order. this reduces nondeterminism but can restrain flexibility/performant implementations.

in SKI: normal order is to always do leftmost spine. (lazy evaluation) in practice call-by value is more popular.

Confluence

no matter what order of evaluation you choose, result does not change

One step diamond property -> confluent by induction.

try to convert problem into equivalent problem that satisfies one step diamond property

Combinator calculus has no variables. only way to pass variables around is to duplicate it using S and pass it to everywhere in the subtree that needs it. not well suited to hardware.

advantage is that it is suited to parallel and distributed systems.

Lecture 4. Lambda Calculus

Lambda function $\lambda x.\, e$ is a function definition. anonymous function.

body of lambda abstraction extends as far right as possible. to the end of the expression or an unmatched right paren.

Rule: The body of a lambda abstraction extends as far right as possible.
to the end of the expression or an unmatched right paren

λx.x λy.y = λx.(x λy.y)

λx.(λy.λz.y z) x is different from λx.λy.λz.y z x = λx.λy.λz.(y z x)

Rule: Application associates to the left

So f x y z = ((f x) y) z

Computation rule: In a function call, the formal parameter x is replaced by the actual argument e_2 in the body of the function e_1

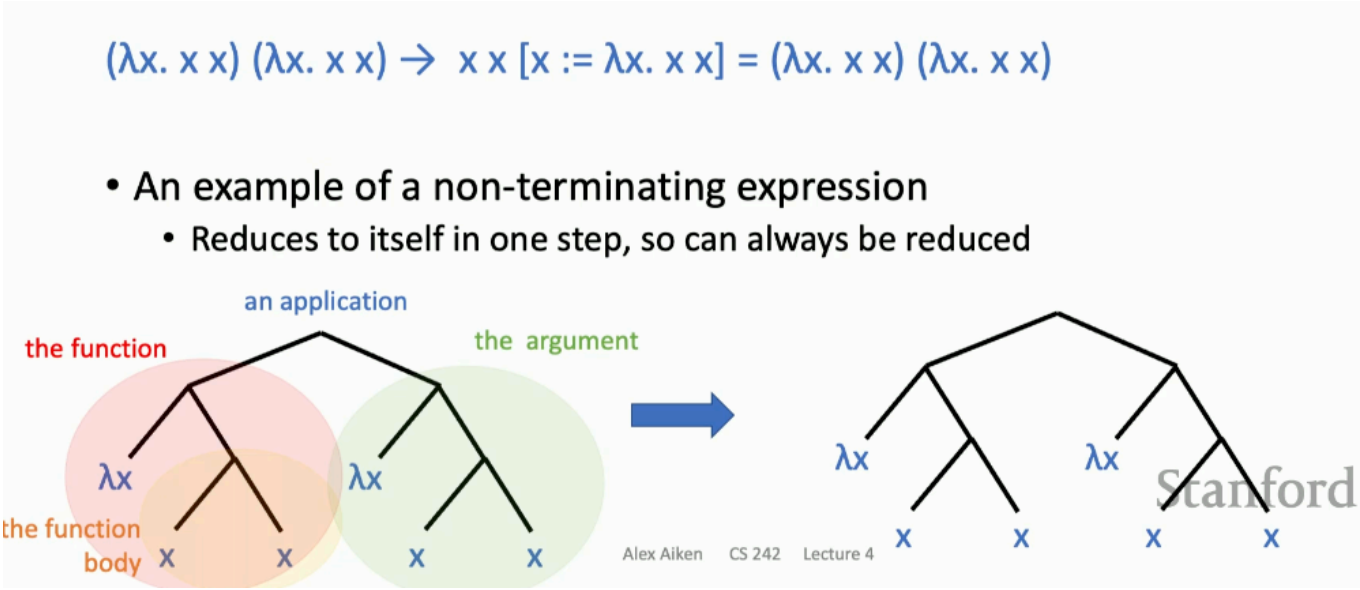
Examples:
identity I: $\lambda x.\, x$
constant function K: $\lambda z.\, \lambda y.\, z$

Substitution is defined recursively down the tree. some complications with substitution that arise with variable name collisions.

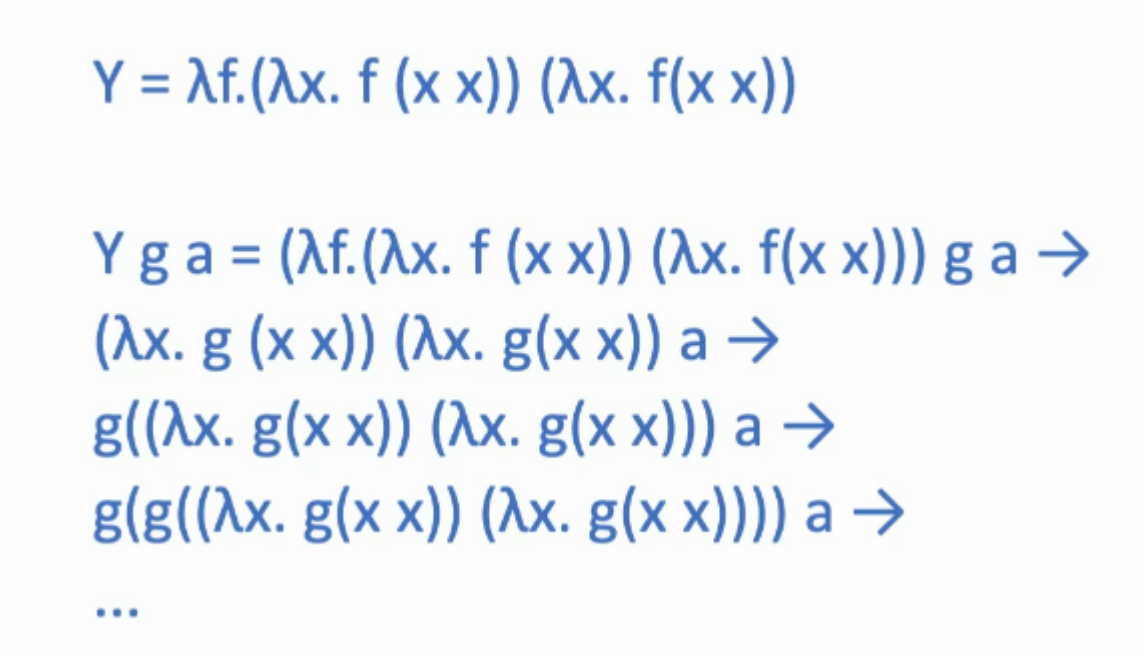
Free variables of an expression are the variables not bound in an abstraction.

To fix this substitution issue, we can rename the lambda y variable to a new variable z to avoid name collisions. (alpha conversion)

Nonterminating expression:



Recursion: Y combinator!



Booleans

We can define booleans as combinators, same as in SKI.

True = $\lambda x \lambda y.\, x$
False = $\lambda x \lambda y.\, y$

Natural Numbers

- n applies its first argument n times to its second argument

$n\ f\ x = f^n(x)$

$0\ f\ x = x$ so $0 = \lambda f.\lambda x.x$

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

Factorial

Factorial

$one = succ\ 0$
$add = \lambda m.\lambda n.\ m\ succ\ n$
$mul = \lambda m.\lambda n.\ m\ (add\ n)\ 0$

$pair = \lambda a.\lambda b.\lambda f.\ f\ a\ b$
$fst = \lambda x.\lambda y.x$
$snd = \lambda x.\lambda y.y$

$p = \lambda p.\ pair\ (mul\ (p\ fst)\ (p\ snd))\ (succ\ (p\ snd))$
$! = \lambda n.(n\ p\ (pair\ one\ one)\ fst)$

Algebraic Data Types

Union of multiple data types, with constructor, can be recursively defined. has constructors and deconstructors. holds data and gives data.

Natural numbers are an example of an ADT.

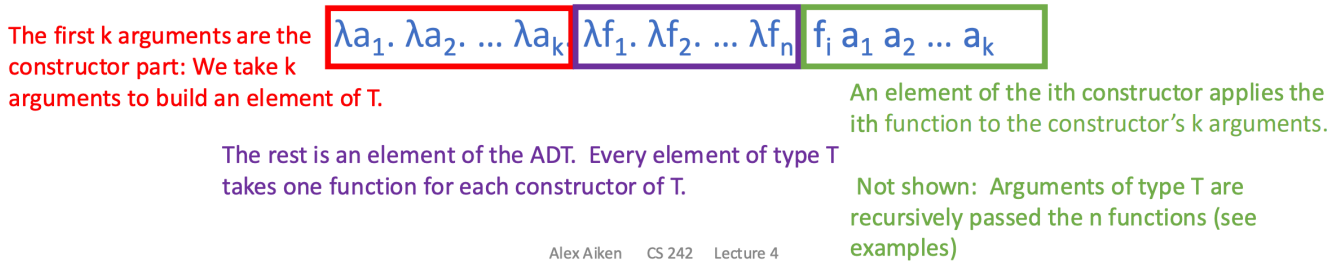
Type Nat = succ Nat |
0

Type List is nil or cons Nat List

Type Tree = left Nat | branch Tree Tree

Encoding Algebraic Types in Lambda Calculus:

ADT T with n constructors. ith constructor C_i has k arguments. Then recursively apply the destructors to each argument



Reduction order

- Lambda calculus is confluent
- normal order: same as SKI, leftmost, lazy - but guaranteed to terminate if it should terminate
- call-by-value: recursively evaluate argument before function application
  - more efficient, evaluates arguments one time

Lecture 5. Type Systems

Type is a set of values. `List` is a function from types to types. `type -> type` is the set of functions mapping `type` to `type`.

Convention is to write hypotheses above conclusions. Also add $\vdash$ in front of each to mean "it is provable that..." because some things are true but not provable given a language.

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \text{ [Add]}$$

Parameterized templates for proofs.

Type Environments

A is a function from variables to Types. $A \vdash e : \text{int}$ means that under assumptions of type environment A, e has type int.

Simply Typed Lambda Calculus

Same as untyped lambda calculus, but we now have to provide types for each argument. For example, $\lambda x.\, e$ becomes $\lambda x : t.\, e$

when we have chained type rules, association is to the right: $a \to b \to a$ means $a \to (b \to a)$

Recursive types are not allowed in simply-typed lambda calc. For example where $\lambda x : ?.\, xx$, x must be both a function type that takes in its own type.

- guarantees system will terminate

Type Rules

$$\frac{}{A, x : t \vdash x : t} \text{ [Var]} \qquad \frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \text{ [Abs]}$$

$$\frac{}{A \vdash i : \text{int}} \text{ [Int]} \qquad \frac{A \vdash e_1 : t \to t' \quad A \vdash e_2 : t}{A \vdash e_1\, e_2 : t'} \text{ [App]}$$

Problems with Type systems:

- code duplication may be required to type programs. tedious
- programmer may required to write in lots of types, verbose

Type Inference

- instead of explicitly writing types, have program infer types.
- for rules, every place a type is required, a fresh type variable is used (stands for definite but unknown type)
- to construct a valid typing, solve equations of typing then substitute back into type derivation to obtain valid proof.

Constraints:

- symmetry, transitivity, structure (functions equal implies inputs and outputs equal)
- adding new constraints, keeping old ones. eventually will add all implications of original constraints

Solutions:

- if function type is equal to int, no solution.
- if recursive / infinite, no finite solutions.

Canonicalization

Algorithm to confirm that saturated type constraints are valid. set of rules to recursively check the relations. Assigns each type variable to a canonical type

Given a saturated set of equations S and a type t, the canonicalization algorithm $C(\emptyset,S,t)$ produces a canonical type for t that does not depend on S

$C(X, S, int) = int$

$C(X, S, t \to t') = C(X \cup \{t\}, S, t) \to C(X \cup \{t'\}, S, t')$ if $t \notin X, t' \notin X$

$C(X, S, \alpha) = C(X \cup \{t\}, S, t)$ if $\alpha = t \in S$, t is not a type variable, $t \notin X$

$C(X, S, \alpha) = C(X \cup \{\alpha\}, S, \beta)$ if $\alpha = \beta \in S, \alpha < \beta, \alpha \notin X$

$C(X, S, \alpha) = \alpha$ otherwise

Lecture 6. Polymorphic Types

Added quantifiers outside of type. types cannot have quantifiers as part of them, but quantifiers can describe types.

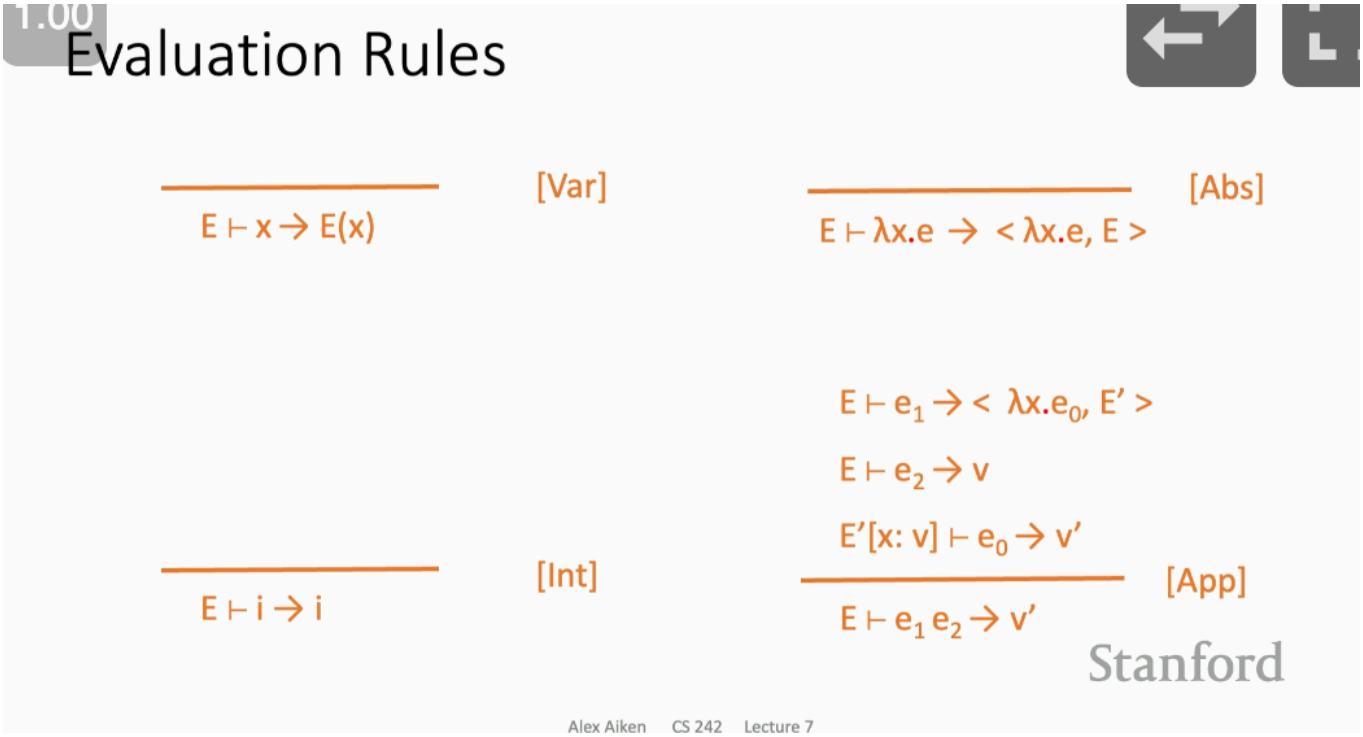
when quantifier describes a function type, we can use that function on different typed inputs because we never specify the type explicitly (as opposed to needing to first duplicate the function application through rewriting)

To do type inference on polymorphic types, do each subtree above a let, with normal type checking. then recursively go down the tree.

this polymorphic type inference / checking system powers most modern languages: functional and non functional.

Lecture 7: State

Function evaluation rules



Abs binds the free variables in an environment to the function. That way when the function is used elsewhere, it uses the same free variables. Lexical, form a closure.

Lambda Calculus with States

$e \rightarrow x \mid \lambda x.e \mid e\ e \mid i \mid new \mid !e \mid e := e$

new	allocate a new memory location x and return a pointer to x (initialize x to 0)
!e	dereference a pointer
$e_1 := e_2$	assign through a pointer, value is e_2

Store

- mapping from memory locations to values
- locations are values

New

- returns a new memory location that is unused in the store
- sets value at that store to be zero

Lecture 8: Continuations

```
let x = e in
  let y = e' in
    let z = x + y in
      z
```

Can be read as a sequential program

```
x = e
y = e'
z = x + y
```

Order of evaluation is explicit
Every intermediate result has a name

A continuation is a function that takes a value as an argument and evaluates the rest of the program. Just performs one primitive step of computation

$$C(x,k) = k\ x$$
$$C(\lambda x.e,\ k) = k\ (\lambda k'.\lambda x.\ C(e,k'))$$
$$C(e\ e',k) = C(e,\ \lambda f.C(e',\ \lambda v.f\ k\ v))$$
$$C(i,k) = k\ i$$
$$C(e + e',k) = \ C(e,\ \lambda v.C(e',\ \lambda v'.k\ (v + v'))$$

AnExample

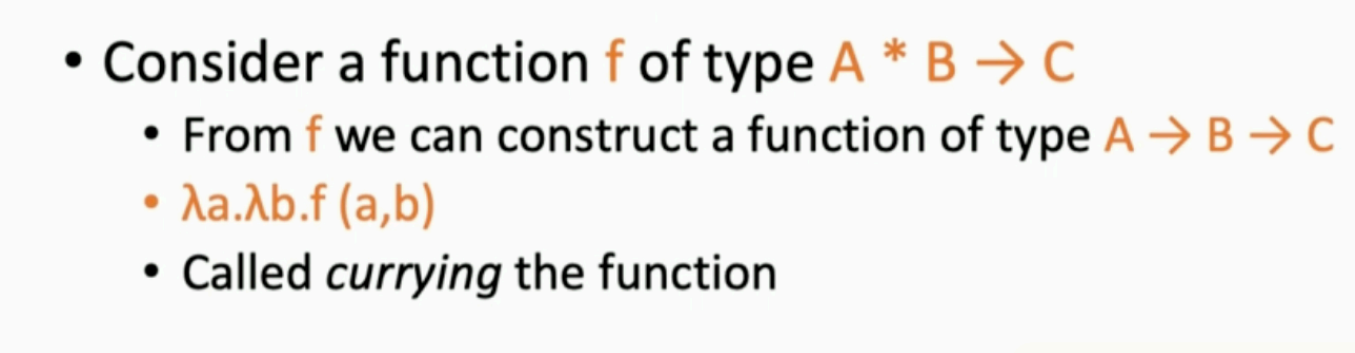
$$C((\lambda x.x + 1)\ 2,\ k_0) =$$

$$C(\lambda x.x + 1,\ \lambda f.C(2,\ \lambda v_0.f\ k_0\ v_0)) =$$
$$C(\lambda x.x + 1,\ \lambda f.((\lambda v_0.f\ k_0\ v_0)\ 2)) =$$
$$(\lambda f.((\lambda v_0.f\ k_0\ v_0)\ 2))\ \lambda k_1.\lambda x.C(x + 1,k_1) =$$
$$(\lambda f.((\lambda v_0.f\ k_0\ v_0)\ 2))\ \lambda k_1.\lambda x.C(x,\ \lambda v_1.C(1,\ \lambda v_2.\ k_1\ (v_1+v_2))) =$$
$$(\lambda f.((\lambda v_0.f\ k_0\ v_0)\ 2))\ \lambda k_1.\lambda x.C(x,\ \lambda v_1.(\lambda v_2.\ k_1\ (v_1+v_2))\ 1) =$$
$$(\lambda f.((\lambda v_0.f\ k_0\ v_0)\ 2))\ \lambda k_1.\lambda x.(\lambda v_1.(\lambda v_2.\ k_1\ (v_1+v_2))\ 1)\ x$$

Review continuations... kinda went over my head..

Lecture 9: Monads

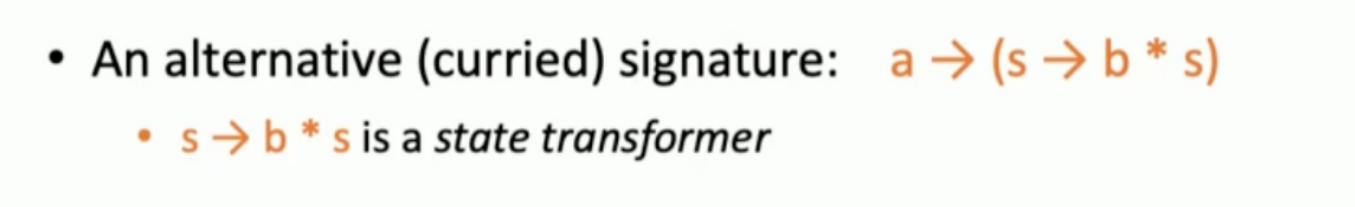
Currying: abstracting a function that takes in a pair into a function that takes in its individual parts



Exceptions:
Should be strict - whenever an expression evaluates to or sees an Exception, should return Exception.

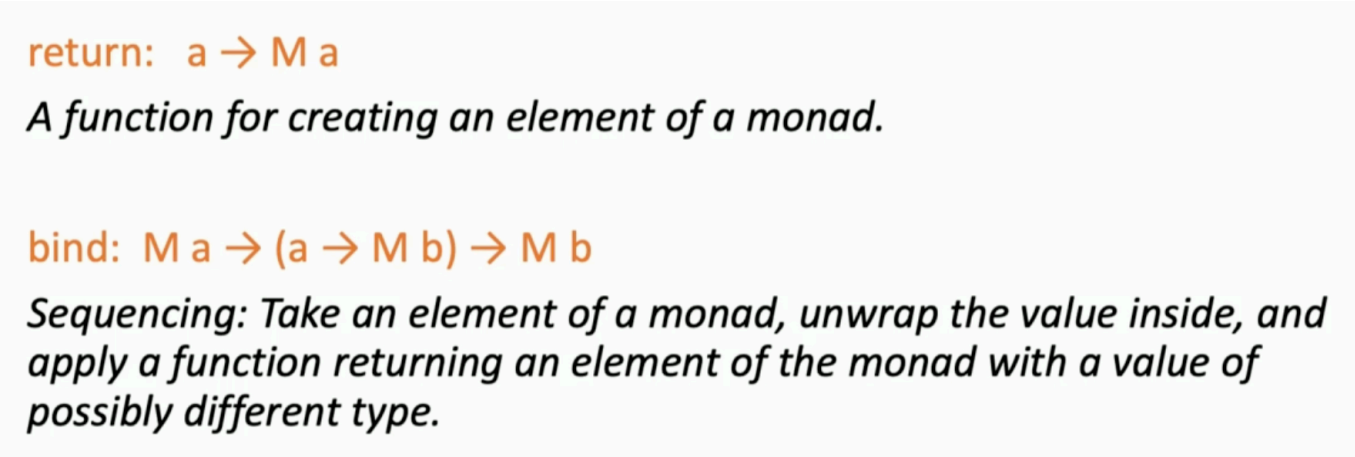
- Propagation of exception

Why can't we just do states in pure lambda calc? We could represent state as part of the type



State Monad:
Monads M b = s -> b -> s', a delayed application of b to a state. Basically a pending program that doesn't execute until an initial state is supplied, thus generating an end state

More generally Monad wraps an expression and allows binding of Monads. Must be defined for each language feature. Casework is captured within the monad definition.



Lecture 10: Objects

Records

set of field value pairs, order doesn't matter

Records with a fixed set of fields are special case of ADTs

Objects

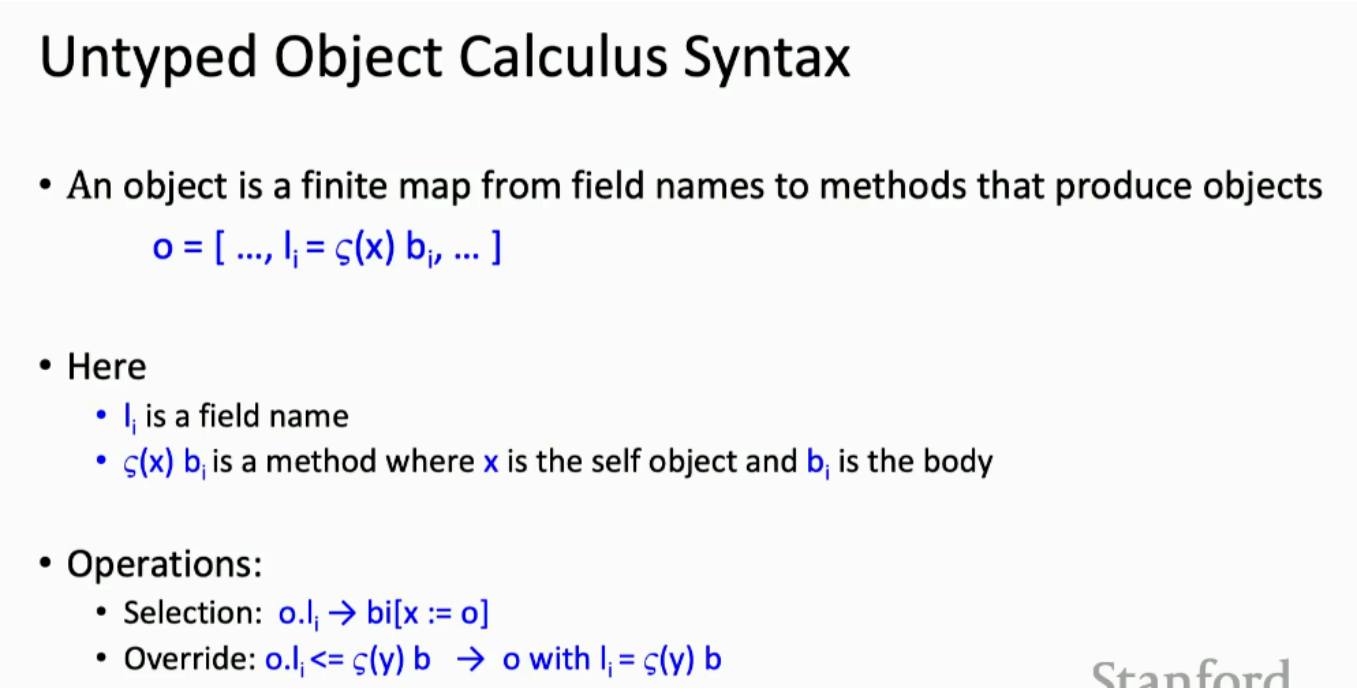
Objects are a collection of fields and method son those fields. Can represent the fields with record.

For methods, we can add method fields to the record, which are just functions.

- Each method implicitly takes in the object itself, so actually the types of the methods are functions from type X (object record type) to the output of the method.
- Thus, objects have recursive definitions

Object Calculus

Not lambda calc, new system



Don't need to distinguish fields and methods. fields are just constant methods

Can programmatically override a method. - this allows changing fields but also changing methods.

Backup Methods

$o = [\ retrieve = \varsigma(x)\ x,$
$\quad backup = \varsigma(x)\ x.retrieve <= \varsigma(y)\ x\ ]$

Every time the backup method is called, it modifies the retrieve method to return the self object at the time the backup method was invoked.

Can create backup by storing current version of object in backup

Natural Numbers

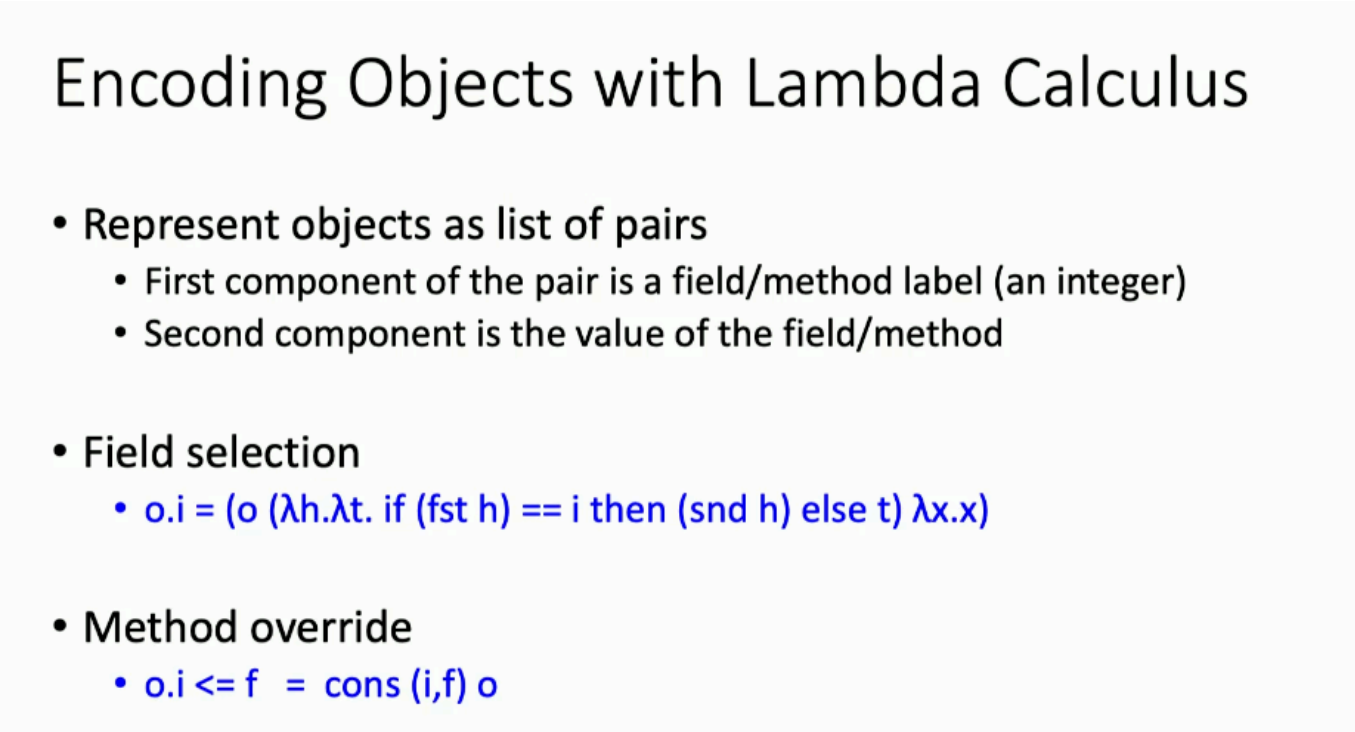
$zero = [\ \ iszero = \varsigma(x)\ true,$
$\quad pred = \varsigma(x)\ x,$
$\quad succ = \varsigma(x)\ (x.iszero <= \varsigma(y)\ false).pred <= \varsigma(y)\ x\ ]$

Stores previous number object in pred. basically a stack of objects on top of a zero.

Encoding Lambda Calculus with Objects

$T(x) = x$
$T(e_1\ e_2) = (T(e_1).arg <= \varsigma(y)\ T(e_2)).val$
$T(\lambda x.e) = [arg = \varsigma(x)\ x.arg,\ val = \varsigma(x)T(e)[x := x.arg]\ ]$

Encodes lambda calculus expressions as objects.



Untyped object systems and untyped lambda calc are easily converted between, so they are equivalent in computational power.

But same is not true for typed version because of the override functionality. There is no way to generate constraints to account for arbitrary change of method via averride.

Typed Object-oriented languages

Type object-oriented languages is subtyping. Basically like when we have a type that can be used anywhere another type can be used, which means that A has at least all the fields in B if A < B.

Solution 1. Mainstream typed OO

- restrict the definition of methods to a first phase before methods are typed.
- inheritance, static override, restrictions on modifying superclasses, dynamic update only of fields.
- guarantees assembly of object's type is independent of program evaluation
- e.g. C++, Java

Solution 2. Functional + OO

- add OO features to lambda calculus.
- functional language does most of the work, OO extensions are thin.
- Every functional language has a object system

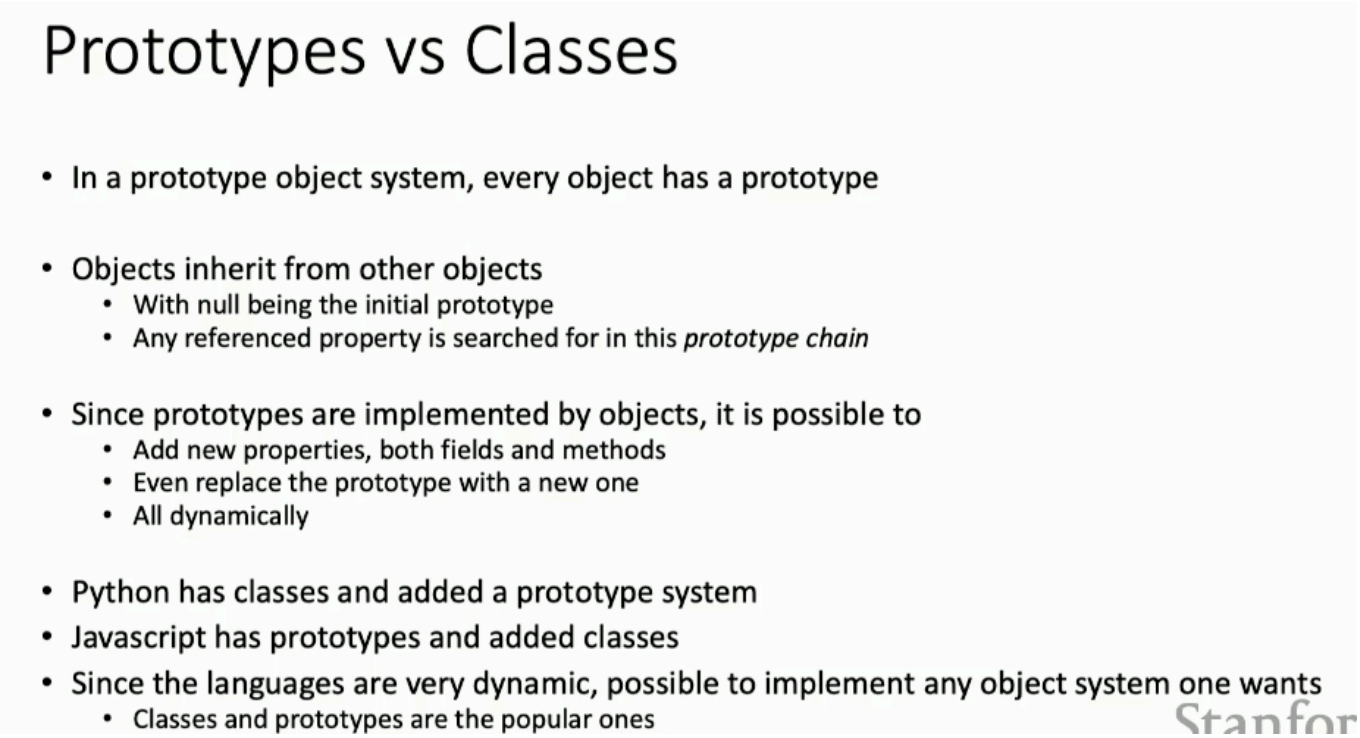
- e.g OCaml, Haskell

Solution 3. OO + Functional

- adding functional features to an OO language
- adding first-class functions, parametric polymorphism (templating)

Solution 4. Dynamically Typed

- give up on static typing
- e.g. JavaScript, Python
- Uses prototype systems



Lecture 11: Rust

Memory safety

- property that pointers point to objects in memory
- causes many memory errors that go uncaught

How to ensure safety?

- automatically via GC
- systematic but unenforced programming practices
- automatically via a static type system

Garbage Collection

- deallocation is done automatically, not by the programmer
- no pointer arithmetic allowed
- indexing into arrays is bounds checked
- memory-safe!
  Downsides:
- performance costs
- expensive bounds checks
- unpredictable delays / inefficient for memory intensive applications

Ownership programming

- diesgners of large systems have always needed to talk about system's rules for memory management. (who is responsible for deallocating)
- give ptrs owners!
- every object/memory block has a unique owner

Aliasing:

- two ptrs to same object in memory
- need to coordinate read or read/write, and make sure deallocating is only done when no other aliases of the ptr exist

In C, a restricted pointer cannot be aliased by another pointer (not a guarantee)
OO code is particularly prone to generating aliasing

Idea #1: Disallow mutation

aliasing is only a problem when used in conjunction with mutation
most things can be done, but big problem is in-place update of array (O(1) vs O(logn))

Practical approach: separate mutable and immutable

immutable objects as default, need to specifically declare mutable objects.

Idea #2: Control aliasing

Ownership types, compiler tracks aliasing

Rust!

there is always a single owner reference of every object

- Owner is responsible for the resources of the object.
- transfers ownership open assignment of variable - cannot use original ptr afterwards

Lifetimes:

- between when ptr is owner to when it goes out of scope. each object cannot have overlapping lifetimes

Can create explicit aliases, called *borrows*

- mutable and immutable borrows

Borrows

- borrow cannot outlive its owner
- borrow cannot deallocate its object
- they can only be one mutable borrow to an object in scope

lifetime of output needs to be available at compile time - so we need to annotate

Lecture 12: Logic Programming

A proof is constructive if it provides a way to exhibit this function. e.g. for every list of integers, there is a list y with the same elements arranged in non-descending order. A constructive proof for this is an algorithm for sorting lists of integers.

Proof by contradiction is never constructive.

PROLOG

- PROgramming in LOGic
- automated theorem prover

PROLOG can return whether a predicate is true if arguments provided. Otherwise, it will return whether a argument can be filled in to satisfy the predicate, and which argument. This is basically a function. it basically tries to fill free variables to satisfy the predicate

PROLOG has terms and atoms

- term: constant, variable or constructor
- atom: predicate applied to terms

PROLOG program has facts and rules

- fact is a rule with no rhs. always true.
- rule: if all predicates on rhs are true, lhs is true. (lhs is true if rhs is true)

Semantics

- Let σ range over all *ground substitutions*
  - Substitutions that map variables to terms with no variables in them
- Given a set of rules

$$P_1(t_{11},\ldots) :- P_2(t_{21},\ldots), \ldots, P_n(t_{n1},\ldots)$$

- The semantics is the smallest set of atoms *F* satisfying

$$\{\sigma(P_2(t_{21},\ldots)), \ldots, \sigma(P_n(t_{n1},\ldots))\} \subseteq F \Rightarrow \sigma(P_1(t_{11},\ldots)) \in F$$

Basically, for all grounded inputs of the rules, the semantics is the smallest set of atoms F that is closed under those rules.

Implementations

implementations do not follow semantics, since efficiency is a major problem in logic languages

PROLOG takes goals and works backwards towards facts. Matches goals to rules and recursively adds new subgoals, until all goals have been satisfied. Can backtrack through matching rules until one is satisfied, or none are satisfied.

in other words, PROLOG does proof search to find a proof tree starting from the goal.

Semantics implies BFS of the goal, but this is really slow. in practice, we do DFS, which can lead to non-termination.

Substitutions

- *Revised rule:* To satisfy a goal g, find the first untried rule G :- H1,…,Hn such that s1 = unify(g,G)
  - unify computes a substitution s1 such that s1(g) = s1(G)
  - Add s1(H1) as a subgoal.
  - If s1(H1) succeeds, it returns a substitution s2
  - Add s2(s1(H2)) as a subgoal, repeat.
  - If all subgoals succeed, result is the substitution sn o … o s2 o s1

Stanford

Backtracking, we now need to consider same predicates and rules with different substitutions.

Cut

- Backtracking can be expensive, so PROLOG includes a feature ! (pronounced "cut") to control it

- Consider A :- B, C, !, D
  - PROLOG will not backtrack past a !
  - If D fails, the implementation will not attempt to resatisfy B and C
  - The entire rhs fails immediately

Opinions

- good for certain applications (search)
- only really has this one built in algo, backtracking search
- not scalable
- usually untyped or weakly typed

- But special-purpose logic programming is commercially important
  - Domain-specific logic languages for scheduling
    - airline crews, trucking, manufacturing, chip design
    - Use search techniques and constraint languages to solve NP-hard problems
  - Databases
  - Programming languages
    - Type inference!

Lecture 13: Haskell

- functional, strongly typed
- primes are valid parts of variable names

Underscores can be used for pattern matching (discarding a value)

```
fst :: (a, b) -> a
fst (x, _) = x
```

Lists are with cons and nil:

```
cons ⟶ :
nil ⟶ []
cons 2 (cons 1 nil)
⟶ 2:(1:[])
[2, 1]
```

```
sum :: [Int] -> Int
sum (x:y) = x + sum y
sum [] = 0
```

like ADTs, uses pattern matching

Creating types

```
type IntList = IntCons Int IntList | IntNil

data IntList = IntCons Int IntList | IntNil
```

can apply things infix by surrounding with backticks

can do partial functions (except in certain inputs), but better to do total functions with compound type return (like optional in c++)

Generic types

```
data List a = Cons a (List a)
            | Nil

l :: List String
l = "hello" `Cons` ("world" `Cons` Nil)

data Maybe a = Just a | Nothing

head :: List a -> Maybe a
head (Cons h _) = h
head Nil        = Nothing


head l
> Just "hello"

head (2 `Cons` (1 `Cons` Nil))
> Just 2
```

use ":t" to get type of something

```
finalPrice :: Float -> Float
finalPrice = cardFee . tax . coupon . sale

                    "pointfree style"
                          └── combinators?
                              abstraction?
```

Side effects

Haskell uses lazy semantics (call by name)

- functions cannot easily handle side effects (mutation, printing, user input, fs)
- functions only evaluate to things (like returning)

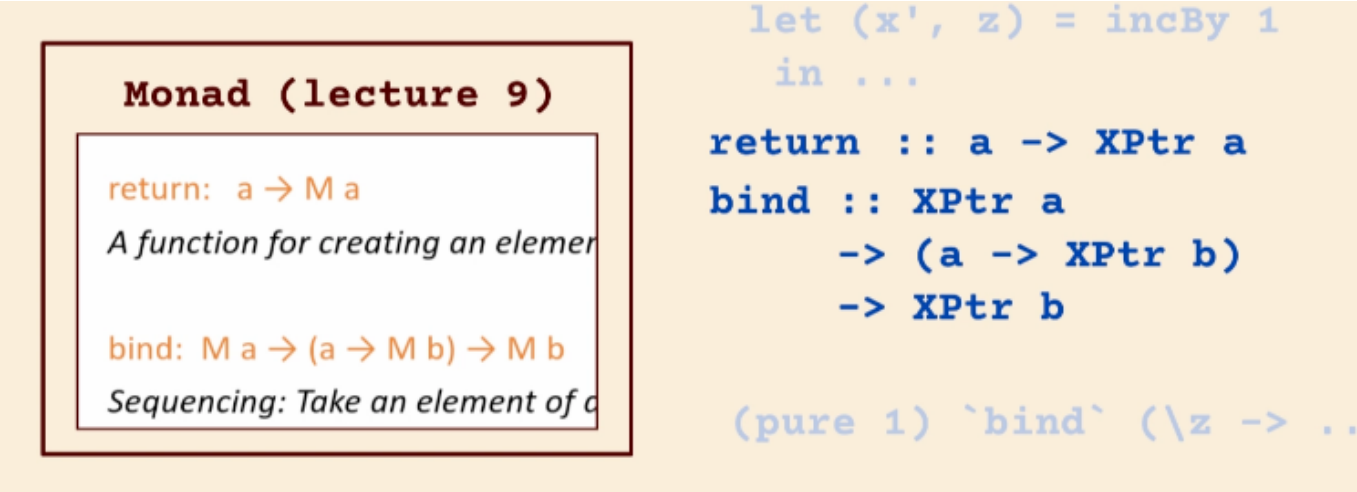
Can either allow side effects (impure language)

- OCaml, Scala, F#
- prevents normal order evaluation
  or forbid all side effects (pure language)
- Haskell

Encoding side effects in Haskell

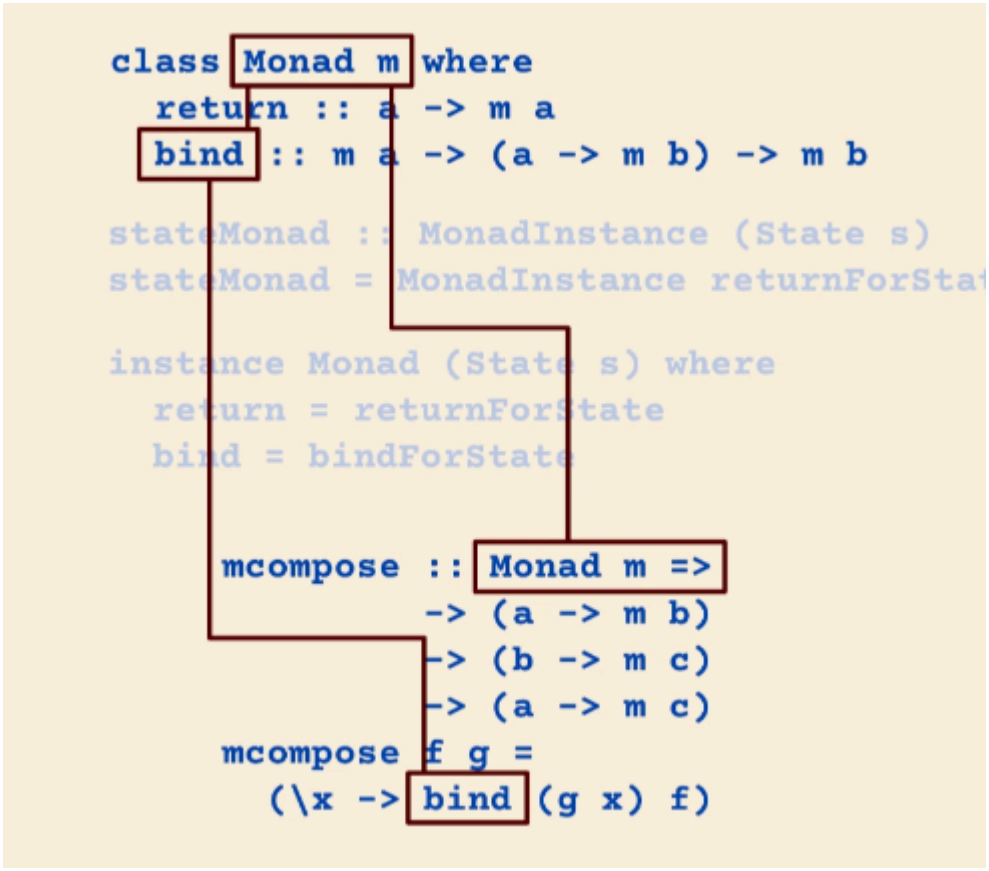
to implement pointers, we can thread integers through all the functions in both input and output. functions can take an additional int input and an additional int output. can express this `int -> (int, return_type)` as `Xptr return_type`.

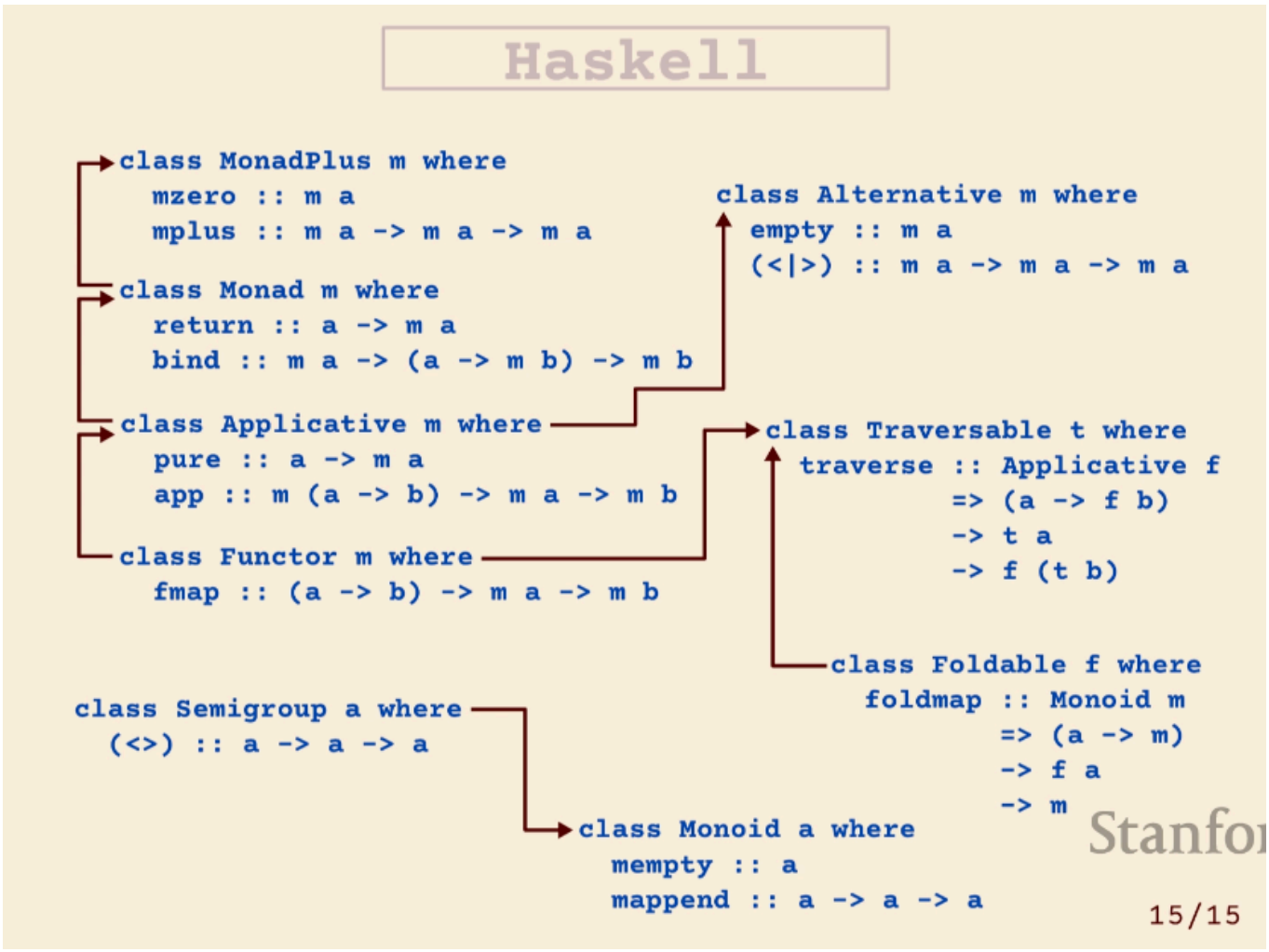
Can define bind as a function to chain function onto XPtr



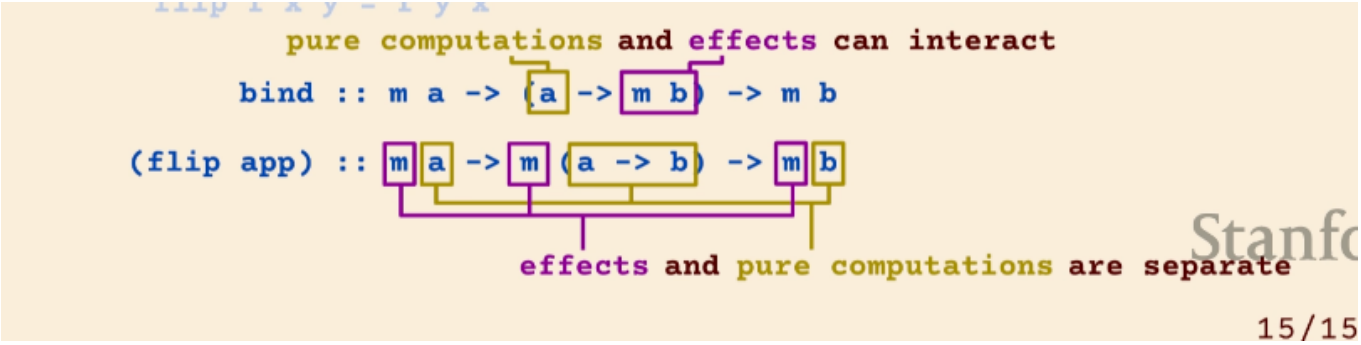
Use monads to encode side effects

Typeclasses allow for pattern matching to ensure something is an instance of a certain typeclass. Compiler automatically figures out which instance of the typeclass we want based on the conditioning.


class Monad m where
return :: a -> m a
bind :: m a -> (a -> m b) -> m b
stateMonad :: MonadInstance (State s)
stateMonad = MonadInstance returnForSta
instance Monad (State s) where
return = returnForState
bind = bindForState
mcompose :: Monad m =>
-> (a -> m b)
-> (b -> m c)
-> (a -> m c)
mcompose f g =
(\x -> bind (g x) f)


Haskell
class MonadPlus m where
mzero :: m a
mplus :: m a -> m a -> m a
class Monad m where
return :: a -> m a
bind :: m a -> (a -> m b) -> m b
class Applicative m where
pure :: a -> m a
app :: m (a -> b) -> m a -> m b
class Functor m where
fmap :: (a -> b) -> m a -> m b
class Alternative m where
empty :: m a
(<|>) :: m a -> m a -> m a
class Traversable t where
traverse :: Applicative f
=> (a -> f b)
-> t a
-> f (t b)
class Foldable f where
foldmap :: Monoid m
=> (a -> m)
-> f a
-> m
class Semigroup a where
(<>) :: a -> a -> a
class Monoid a where
mempty :: a
mappend :: a -> a -> a

Monads vs applicatives:



monad inherently has interplay between pure and effectful computations.

Lecture 14: Program Verification

Not only proving types are consistent, but proving arbitrary aspects of programs (sort function actually sorts the list, no race conditions, etc)

$$\frac{A \vdash e_1 : t \to t' \qquad A \vdash e_2 : t}{A \vdash e_1\, e_2 : t'}\ [\text{App}]$$

From a proof of $t \to t'$ and and a proof of t, we can prove t'.

Implication Elimination (modus ponens)

$$\frac{A, x : t \vdash e : t'}{A \vdash \lambda x.e : t \to t'}$$

If assuming t we can prove t', then we can prove $t \to t'$.

Implication Introduction

isomorphism between expressions/types and proofs/propositions

Classical logic is not constructive proof, ?? In software applications we want to have a constructive proof

In set theory, sets cannot contain themselves

What Does Well-Founded Mean?

- There is no set of all sets
- Instead, there is an infinite hierarchy of stratified sets

- We define ``small'' sets at stratum 0
- The set of all level 0 sets is a stratum 1 set
- The set of all level 1 sets is a stratum 2 set
- ...

- In this way no set can be an element of itself
  - Stratum *n* sets can only contain small sets of stratum *n* and sets of strata less than *n*
- Similar to the definition of ordinals

St

Thus in type systems, we must have a type hierarchy, with ordinary types, then types that are generalized on top of these, etc.

Pi Types

Defining the List data type :

List: Type → Type
Cons: $\Pi\ \alpha$: Type. $\alpha \rightarrow$ List(α) → List(α)
Nil: $\Pi\ \alpha$: Type. List(α)

Polymorphic types are an example of *dependent types:* The type de
a parameter. Note how Π functions like $\forall$.

There is also a corresponding sum type Σ that functions like $\exists$

Lecture 15: Agda

```
data Bool : Set where
  false : Bool
  true : Bool
```

"Set" is the equivalent of "Type" in Lecture 14

"Data" declares algebraic data types

```
not : Bool -> Bool
not true = false
not false = true
```

Not is defined using multiple clauses with a *pattern* describing which argument(s) match each clause

Stanford

defining through pattern matching / defining through cases

whitespace is important in agda - variable names need to have spaces around them

Pattern Matching in Agda

- In general a function can be defined by cases:

```
f pattern1 = rhs1
f pattern2 = rhs2
...
```

- Patterns must be exhaustive
  - Every possible case must be covered
- Patterns must be disjoint
  - The patterns in different clauses cannot overlap in what they can match
  - Other languages allow overlapping patterns in different clauses, requires specifying which pattern will take priority if more than one matches

Stanford

Infix Operators

```
_+_ : Nat -> Nat -> Nat
zero + m = m
succ n + m = succ (n + m)

data List (A : Set) : Set
where
 [] : List A
 _::_ : A -> List A -> List A

zlist = zero :: []
```

- An infix operator *op* is declared using the name *_op_*
- An "_" indicates where the argument will go
- More general than infix! If-then-else can be define as an operator

  if_then_else_

Stanford

Explicit Polymorphism, and implicit arguments make writing easier

Records are like structs. can also have polymorphic records

A Safe List Lookup Function

```
lookup : {A : Set}(xs : List A)(n : Nat) -> isTrue (n < length xs) -> A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (succ n) p = lookup xs n p
```

Lookup takes a list xs and a natural number n and returns the nth element of xs.

This lookup function is *safe* – it only type checks if the list has at least n elements.

Stanford

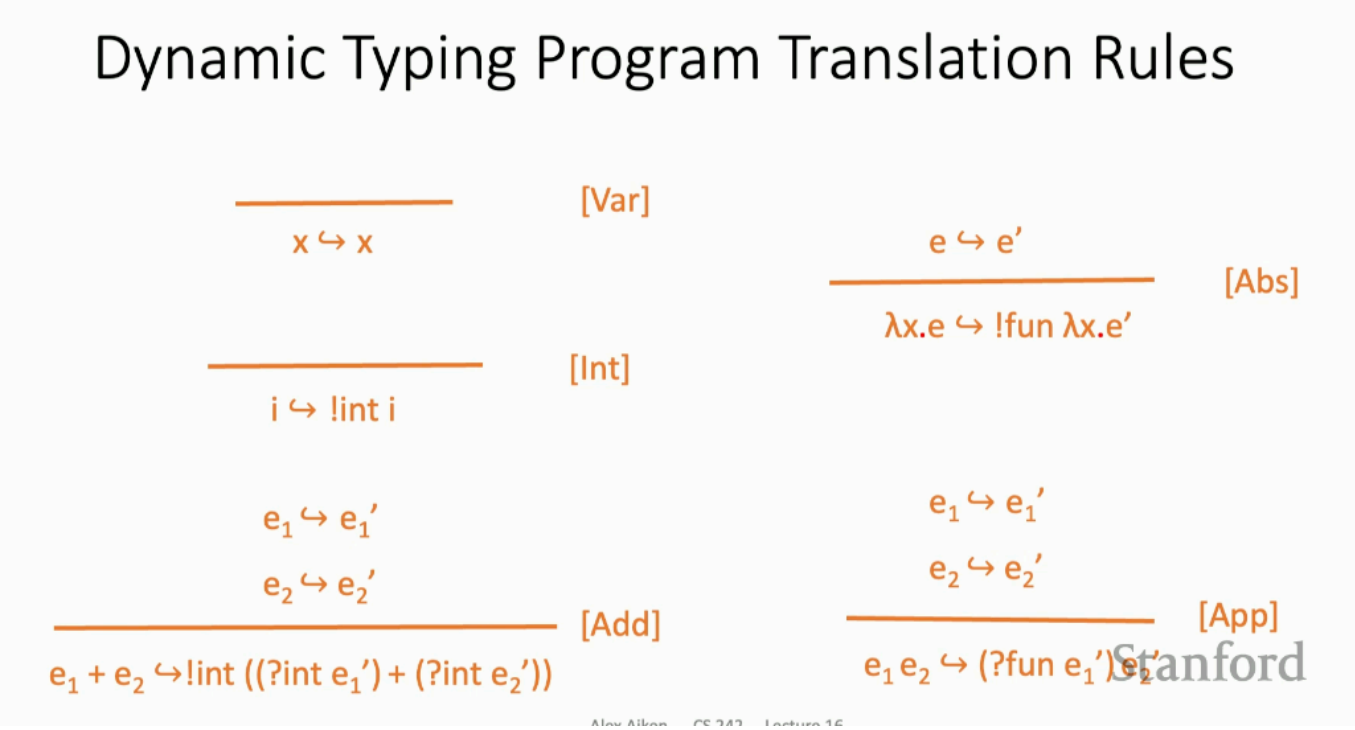
Lecture 16: Gradual Types

Benefits of static typing:

- find type errors when program is written
- faster execution at runtime
- better code , more guarantees

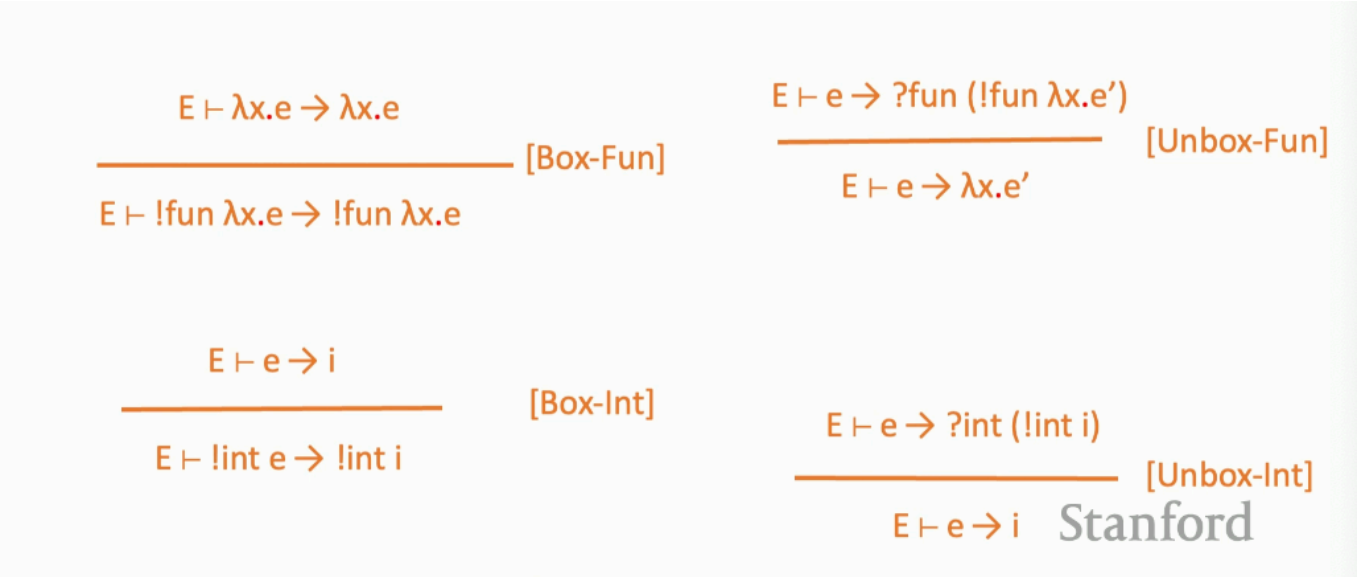
Dynamic typing

- types of checks not at compile type, but at execution time
- type errors will be detected during execution.
- not this is not untyped, which would be if no type checking was enforced.



Dynamic typing is done through tagging and untagging integers

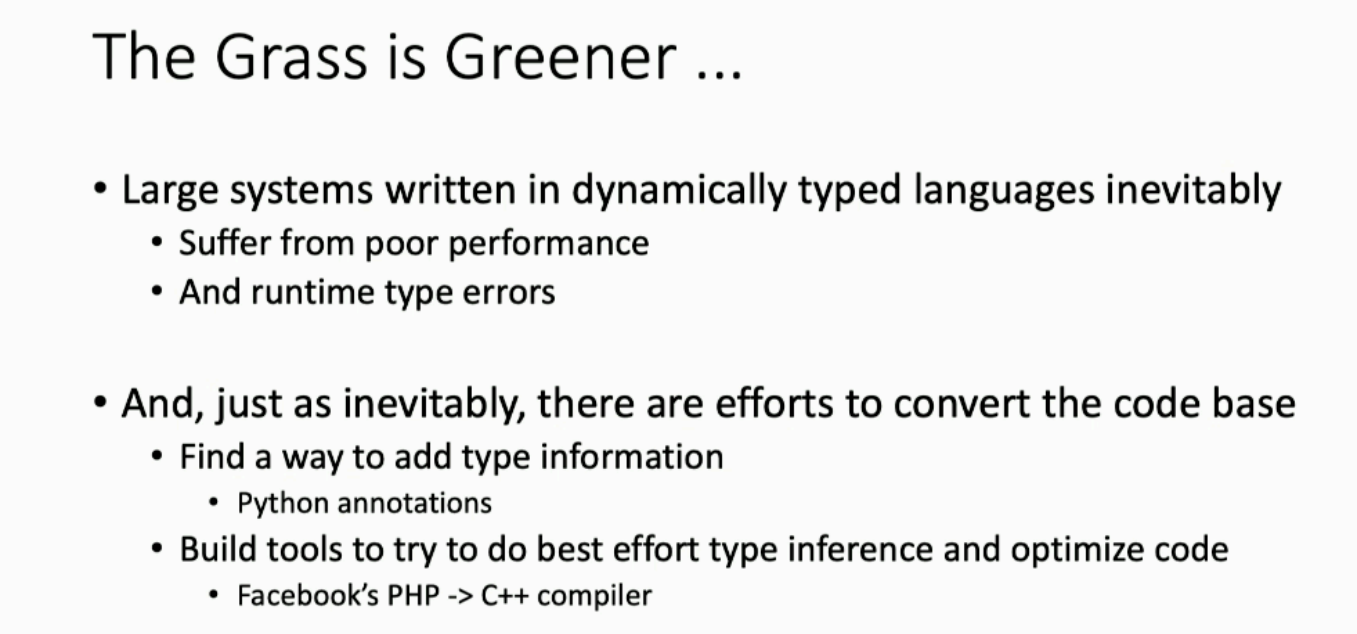
no overhead for user-defined types, only primitive types. user-defined types are already tagged.



Benefits of Dynamic Typing

- guarantees that a specific program execution is type correct
- easy to put together arbitrary code
- lower barrier to entry

Expectation is that statically typed languages should run more important code, and dynamically typed languages for less important code and scripts. But this is not what happened.



Many efforts to try to integrate static type features into dynamically typed languages

MyPy

- allows adding type annotations gradually to Python

- can mix dynamically typed code with "Any" type annotation

Nominal Subtyping

- ColorPoint is a subtype of Point
  - Because ColorPoint explicitly inherits from Point
  - This is *nominal subtyping*
  - By far the most common form of subtyping in practice
- We write ColorPoint $\leq$ Point
  - Anywhere a Point is expected, a ColorPoint can also be used

Structural Subtyping

- Python also supports *structural subtyping*
  - Also called *"duck typing"*
- We consider X $\leq$ Y because X implements the methods of Y
  - No explicit declaration that X inherits Y is needed, the compatibility is determined automatically
- Example: If a class defines a method __iter__ with suitable arguments, MyPy understands it is of type Iterable[T]

Covariant Typing

- Recall: Type constructors are functions that take types as arguments and return types as results
- A type constructor C[A] is covariant if $X \leq Y$ implies $C[X] \leq C[Y]$
- Example: Union[X ,X] $\leq$ Union[X,Y]

List is not covariant - can't always slap on constructor to subtype and expect relationship to be conserved

- not just lists, also for any generic collections of types

Subtyping cannot be used with mutable containers - they must use invariant typing

- in other words, in mutable container types, only subtype if they are equal types

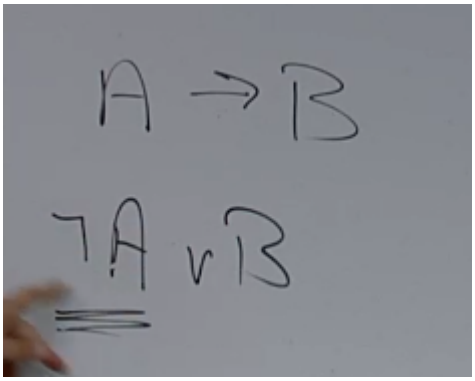
Contravariance

- Function types are *contravariant* in the domain, covariant in the range
- Example: Point $\rightarrow$ Int $\leq$ ColorPoint $\rightarrow$ Int
  - If we expect a function that takes ColorPoint objects as arguments, it is OK to use a function that takes a larger class of arguments such as Point
  - The Point function will just access a subset of the methods of ColorPoint
- In general $A \rightarrow B \leq C \rightarrow D$ if $C \leq A$ and $B \leq D$

Functions are covariant in the range: if we increase the size of the range, we get more functions. makes sense.

Functions are contravariant in the domain. if we increase the size of the domain, we get less functions. Since this is now a more restrictive set of functions that add constraints onto a larger domain.

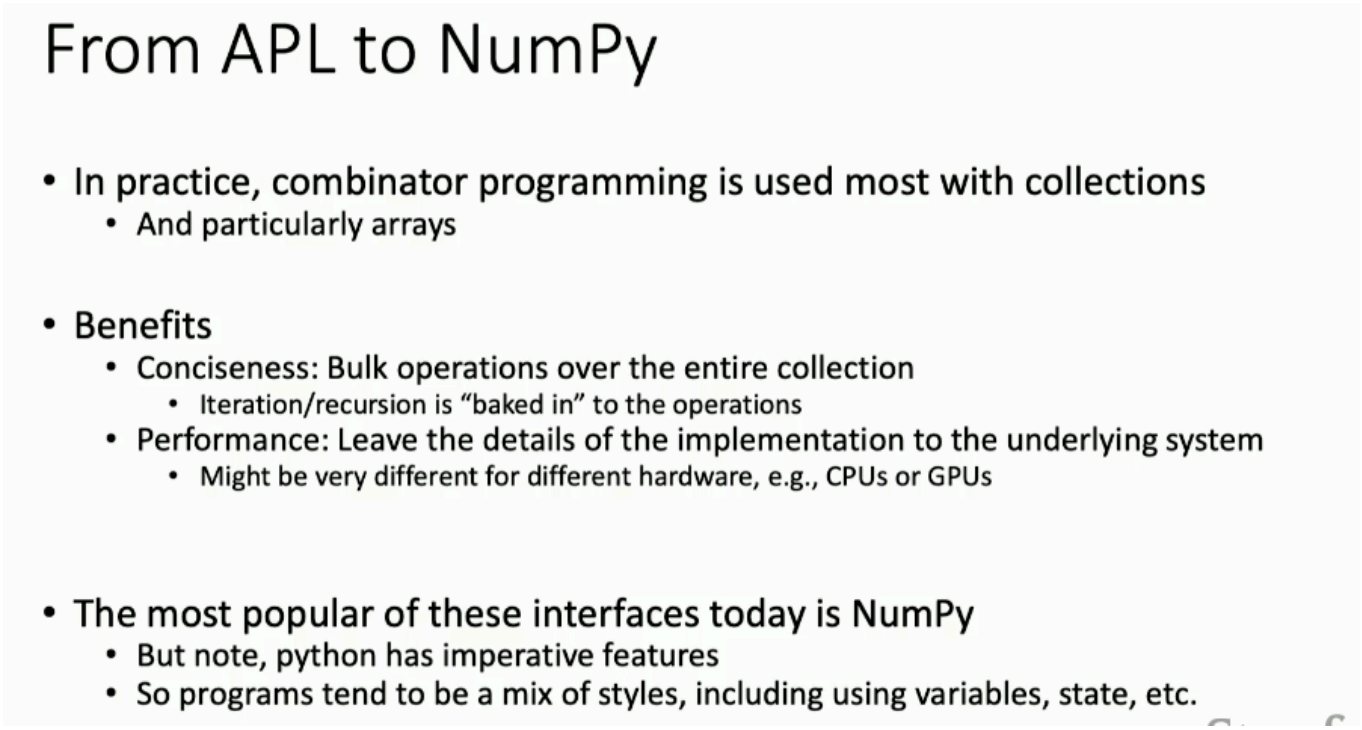
Another way to understand this is with logical isomorphism. We have a function



these are equivalent. Thus, we can see that increasing A will have an inverse effect on the number of such functions, while increasing B increases the number of functions.

Lecture 17: Array Programming

Array programming uses combinator style programming. This is more concise, and also easier to reason about and optimize via compiler.



Broadcasting

- Broadcasting takes two arrays of possibly different dimensions and casts them to arrays of the same dimension
- Rules for broadcast in an array operation A op B
  - If one array has fewer dimensions, add dimensions of size 1 until both have the same number of dimensions
  - For each dimension i
    - If A and B have the same size in dimension i, do nothing
    - If one of A and B has size 1 in dimension i, replicate data in the dimension to the same size as the other array
    - If A and B have different sizes in dimension i and neither is 1, throw an error
- Example
  - A * 5
  - The 5 (a 0-D array) is promoted to a 1-D array of 5's of the same length as A

Lecture 18: Clean up and wrap-up

General Recursion

Recursion

Recall

let $x = e_1$ in e_2 is equivalent to $(\lambda x.e_2)\ e_1$

Extend to recursive definitions

letrec $f = \lambda x.e_1$ in e_2 is equivalent to $(\lambda f.e_2)\ (Y\ \lambda f.\lambda x.e_1)$

recall (lecture 4) that $Y = \lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f(x\ x))$

- diff Y combinator for differ order of evaluations.

Typechecked:

$$\frac{A, f: t_1 \to t_2 \vdash \lambda x.e_1 : t_1 \to t_2 \qquad A, f: t_1 \to t_2 \vdash e_2 : t}{A \vdash \text{letrec } f = \lambda x.e_1 \text{ in } e_2 : t} \text{ [Letrec]}$$

Key difference is that in that first line, we have the definition of f available in our definition of the lambda extraction itself.

To actually find this type, we can just use type inference.

Can't have polymorphic recursion in typesystem

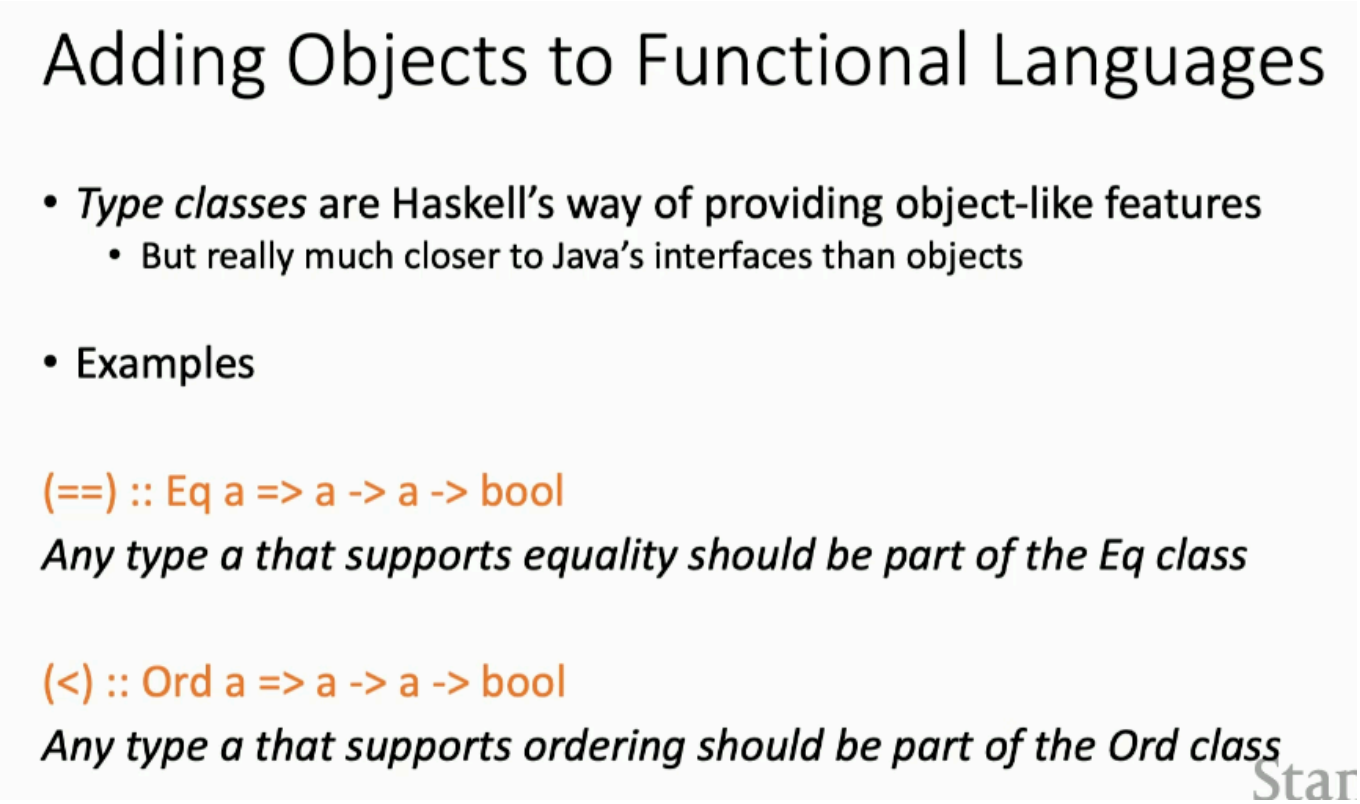
Broadly, this letrec typechecking rule must be built into the system rather than just using Y combinator primitively, since Y Combinator does no typecheck.

Subtyping

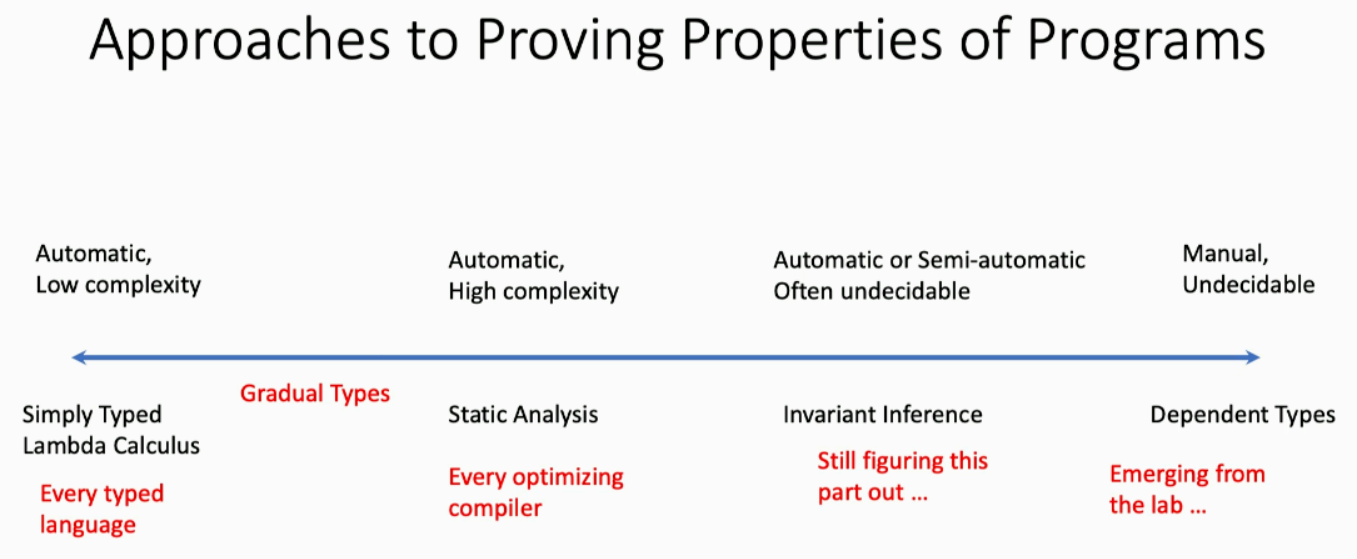
Subtyping: A Subtle Topic

$$\frac{A \vdash e_1 : \text{Bool} \qquad A \vdash e_2 : t_1 \qquad A \vdash e_3 : t_2 \qquad t_1 = t_2}{A \vdash \text{if } e_1\, e_2\, e_3 : t_1} \text{ [If]}$$

$$\frac{A \vdash e_1 : \text{Bool} \qquad A \vdash e_2 : t_1 \qquad A \vdash e_3 : t_2 \qquad t_1 < t \quad t_2 < t}{A \vdash \text{if } e_1\, e_2\, e_3 : t} \text{ [If]}$$



Typeclasses are kinda like virtual classes in C++



Exam Review:

Monads

To think about `bind` (`>>=`) and `return` intuitively in Haskell, it helps to focus on the idea of **values inside contexts** and **chaining computations**. Here's a breakdown:

`return`

Intuition

Think of `return` as a way to **put a value into a context**. It doesn't perform any computation or modify the value — it just wraps it.

Analogy

Imagine putting a gift into a box:

- The gift (value) stays the same.
- The box (context) just contains it.

Key Idea

`return` gives you a value wrapped in a specific monadic context, such as:

- `Just 5` for `Maybe`.
- `[5]` for lists.
- `pure 5` for any `Applicative` monad.

Examples

1. With `Maybe`:

```haskell
return 3 :: Maybe Int
-- Result: Just 3
```

2. With lists:

```haskell
return 3 :: [Int]
-- Result: [3]
```

3. With `IO`:

```haskell
return 3 :: IO Int
-- Result: IO action that returns 3
```

`bind` (`>>=`)

Intuition

`bind` is like a conveyor belt that:

1. **Takes a value out of a context**.
2. **Feeds it into a function** that produces another value in a context.
3. **Chains the resulting context back into the monad**.

Analogy

Imagine you're in a factory:

- A box (context) arrives on a conveyor belt.
- You open the box, process the value inside, and then place the result into a new box.
- The conveyor belt continues with the new box.

Key Idea

`>>=` is all about **sequencing computations** while maintaining the monadic context. It allows you to:

- Extract the value from the first monad.
- Use it in a function that returns another monad.
- Combine these computations into a pipeline.

Examples

1. With `Maybe`:

```
Just 3 >>= (\x -> Just (x + 1))
-- Result: Just 4

Nothing >>= (\x -> Just (x + 1))
-- Result: Nothing
```

   - If the first value is `Nothing`, the entire chain short-circuits.

2. With lists:

```
[1, 2] >>= (\x -> [x, x * 2])
-- Result: [1, 2, 2, 4]
```

- For each value in the list, the function is applied, and the results are concatenated.

3. With `IO`:

```
getLine >>= (\name -> putStrLn ("Hello, " ++ name))
-- Reads a line of input and prints a greeting
```

Comparing `return` and `>>=`

1. **`return`:**

   - Simple: It just wraps a value in the monad's context.
   - Does not perform any computation or modify the value.

   Example:

```
return 5 :: Maybe Int
-- Just 5
```

2. **`>>=`:**

   - Complex: It sequences computations, unwrapping a value from the monad and passing it to the next computation.
   - Maintains the monadic context throughout the chain.

   Example:

```
Just 5 >>= (\x -> Just (x + 1))
-- Just 6
```

Intuitive Pipeline Analogy

You can think of `>>=` and `return` as part of a pipeline:

1. **`return`:** Puts the initial value into a container to start the pipeline.
   - Like boxing up the first item for processing.

2. `>>=` **:** Takes a value out of the container, processes it, and puts the result into a new container.
    - Like opening a box, doing some work, and sending it forward in the chain.

Complete Example:

Let's chain some computations with `Maybe`:

```
computation :: Maybe Int
computation =
  return 5 >>= (\x -> Just (x + 2)) >>= (\y -> if y > 5 then Just y else
Nothing)
```

Steps:

1. `return 5` produces `Just 5`.
2. `Just 5 >>= (\x -> Just (x + 2))` extracts `5`, adds `2`, and produces `Just 7`.
3. `Just 7 >>= (\y -> if y > 5 then Just y else Nothing)` checks the condition and keeps `7` in the monad.

Result: `Just 7`.

Conclusion

- `return` **:** Wraps a value in a monad, starting the chain.
- `>>=` **:** Chains computations while handling the context (e.g., short-circuiting for `Maybe`, concatenating for lists, or sequencing for `IO`).

By thinking of `>>=` as a conveyor belt and `return` as boxing a value, you can build an intuitive understanding of how monads handle computations and contexts.

The `do` notation in Haskell provides a **syntactic sugar** for working with monads. It allows you to write monadic code in a sequential and imperative-like style, making it more readable and easier to understand than chaining `>>=` (bind) calls explicitly.

What Does `do` Do?

1. **Sequencing of Monadic Actions**
   - `do` groups a sequence of monadic actions and executes them in order.
   - Each action can extract values from the monad and pass them to subsequent actions.
2. **Intuitive Syntax**
   - Instead of writing explicit `>>=` and lambdas, `do` makes the code look like a sequence of instructions.
3. **Resulting Monadic Context**
   - The entire `do` block results in a monadic value.

How `do` Works

Translation to `>>=`

A `do` block is internally transformed into a series of `>>=` and lambdas. For example:

```
do
  x <- action1
  y <- action2 x
  action3 y
```

is equivalent to:

```
action1 >>= \x ->
action2 x >>= \y ->
action3 y
```

Key Syntax in `do`

1. **Binding (`<-`)**:
   - Extracts a value from a monadic context.
   - The variable on the left is bound to the value inside the monad.
2. **Pure Actions**:
   - You can include expressions that evaluate to a monadic value without binding them.
   - Example: `putStrLn "Hello, world!"`
3. **Returning Values**:
   - Use `return` to lift a value back into the monad if needed.

Examples of `do` Notation

1. With `Maybe`

```haskell
example :: Maybe String
example = do
  x <- Just 5
  y <- if x > 3 then Just (x + 2) else Nothing
  return ("The result is: " ++ show y)
```

Equivalent to:

```haskell
example =
  Just 5 >>= \x ->
  if x > 3
    then Just (x + 2) >>= \y ->
         return ("The result is: " ++ show y)
    else Nothing
```

Execution:

- `x <- Just 5`: Binds `x = 5`.
- `y <- if x > 3 then Just (x + 2) else Nothing`: Since `x > 3`, `y = 7`.
- `return ("The result is: " ++ show y)`: Lifts the string `"The result is: 7"` into `Maybe`.

Result: `Just "The result is: 7"`

---

2. With `IO`

```haskell
greet :: IO ()
greet = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

Equivalent to:

```haskell
greet =
  putStrLn "What's your name?" >>
  getLine >>= \name ->
  putStrLn ("Hello, " ++ name ++ "!")
```

Execution:

1. Prints `"What's your name?"`.
2. Reads input from the user and binds it to `name`.
3. Prints a personalized greeting.

---

3. With Lists

```haskell
example :: [(Int, Int)]
example = do
  x <- [1, 2]
  y <- [10, 20]
  return (x, y)
```

Equivalent to:

```haskell
example =
  [1, 2] >>= \x ->
  [10, 20] >>= \y ->
  return (x, y)
```

Execution:

- For every `x` in `[1, 2]`, combine it with every `y` in `[10, 20]`.

Result: `[(1,10), (1,20), (2,10), (2,20)]`

Rules for Using `do`

1. **Always in a Monadic Context**
   - A `do` block is valid only if the computations are part of a monad, such as `Maybe`, `IO`, `[]`, etc.

2. **The Last Action Determines the Result**
    - The final line in a `do` block must return a monadic value, and this determines the overall result of the `do` block.
3. **Mixing Pure Code**
    - You can include pure code inside a `do` block, but it won't interact with the monadic context unless wrapped with `return`.

Why Use `do`?

1. **Readability**:
    - Instead of deeply nested `>>=` calls, you get a flat, easy-to-follow sequence of actions.
2. **Imperative Style**:
    - It feels like writing step-by-step instructions, which can be more intuitive for sequential logic.
3. **Abstraction**:
    - `do` hides the explicit chaining of computations, focusing only on the logic of your program.

Conclusion

The `do` notation is a powerful and intuitive way to work with monads in Haskell. It makes sequential computations easier to write and read, while still preserving the underlying monadic behavior. Whether you're dealing with side effects (`IO`), computations that can fail (`Maybe`), or combinations (`[]`), `do` helps streamline your code.