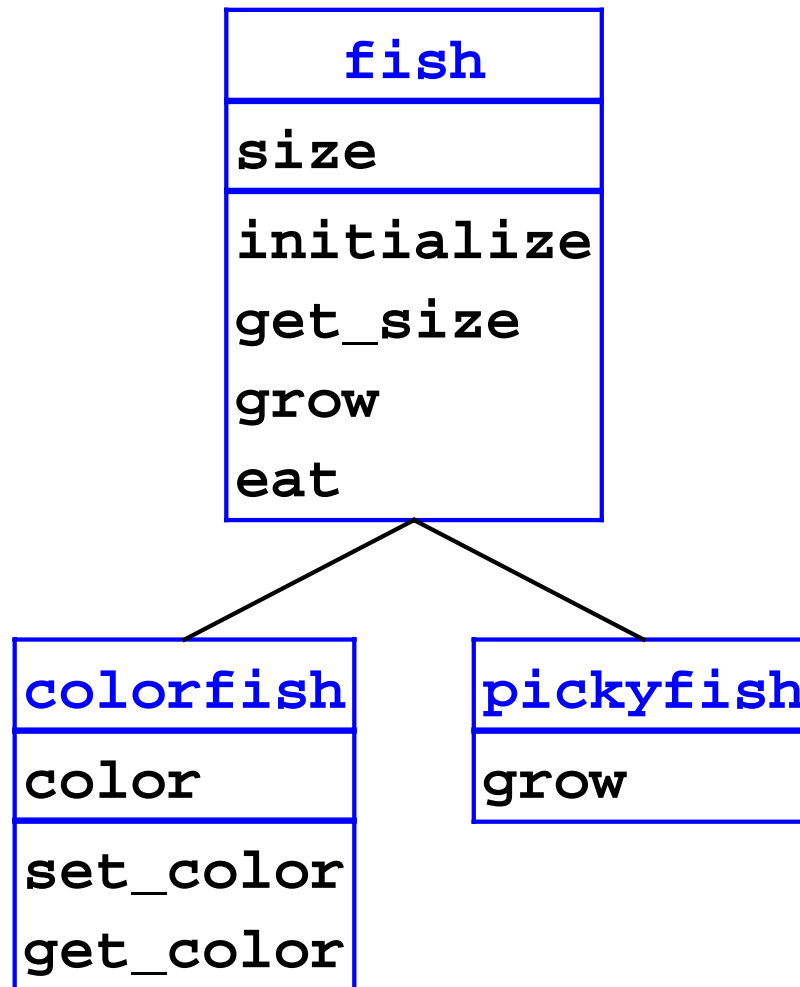


Outline

- More optimizations for our interpreter
- Types for objects

Optimization

Eliminate tree walks: object creation, method calls



Object Creation

Current interpreter:

1. Find class
2. Get field list (walk tree)
3. Allocate field array and object

To eliminate tree walks:

2. Extract flat field list from class

Method Calls

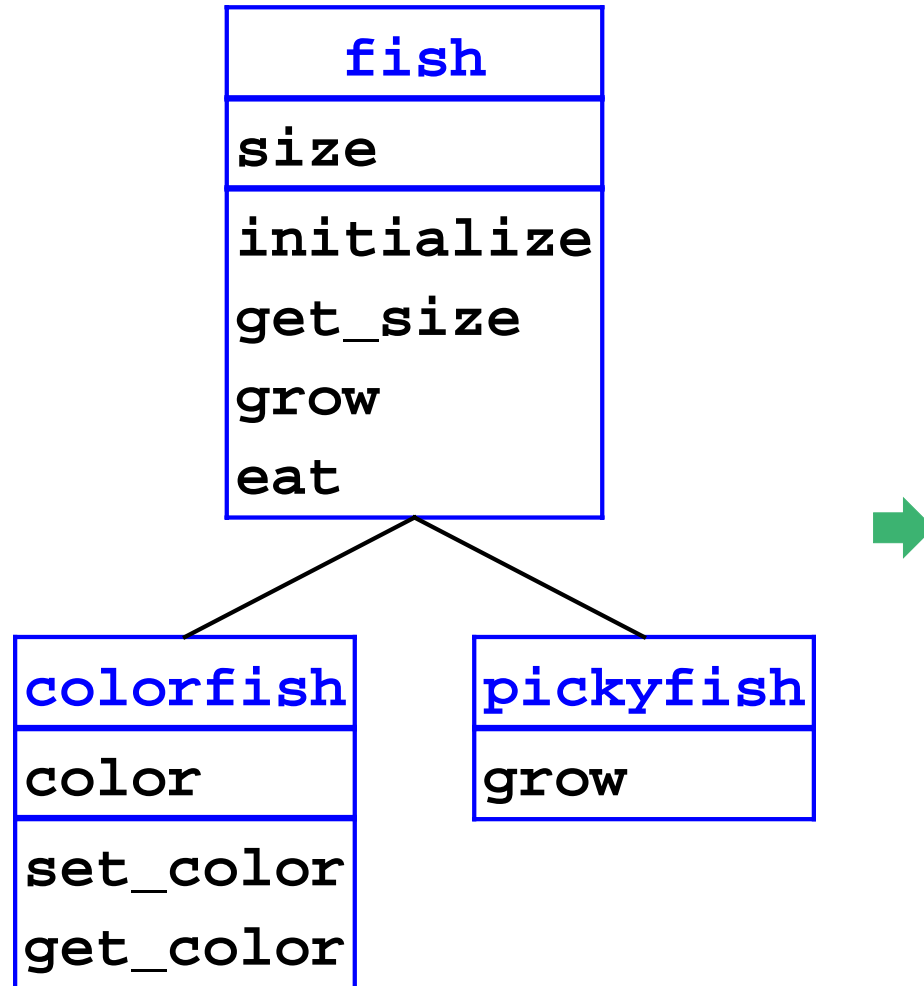
After object and arguments are determined:

1. Lookup object class
2. Find class containing method (walk tree)
3. Get variables for class (walk tree)
4. Create environment: fields + `%super` + `self` + args
5. Evaluate method body

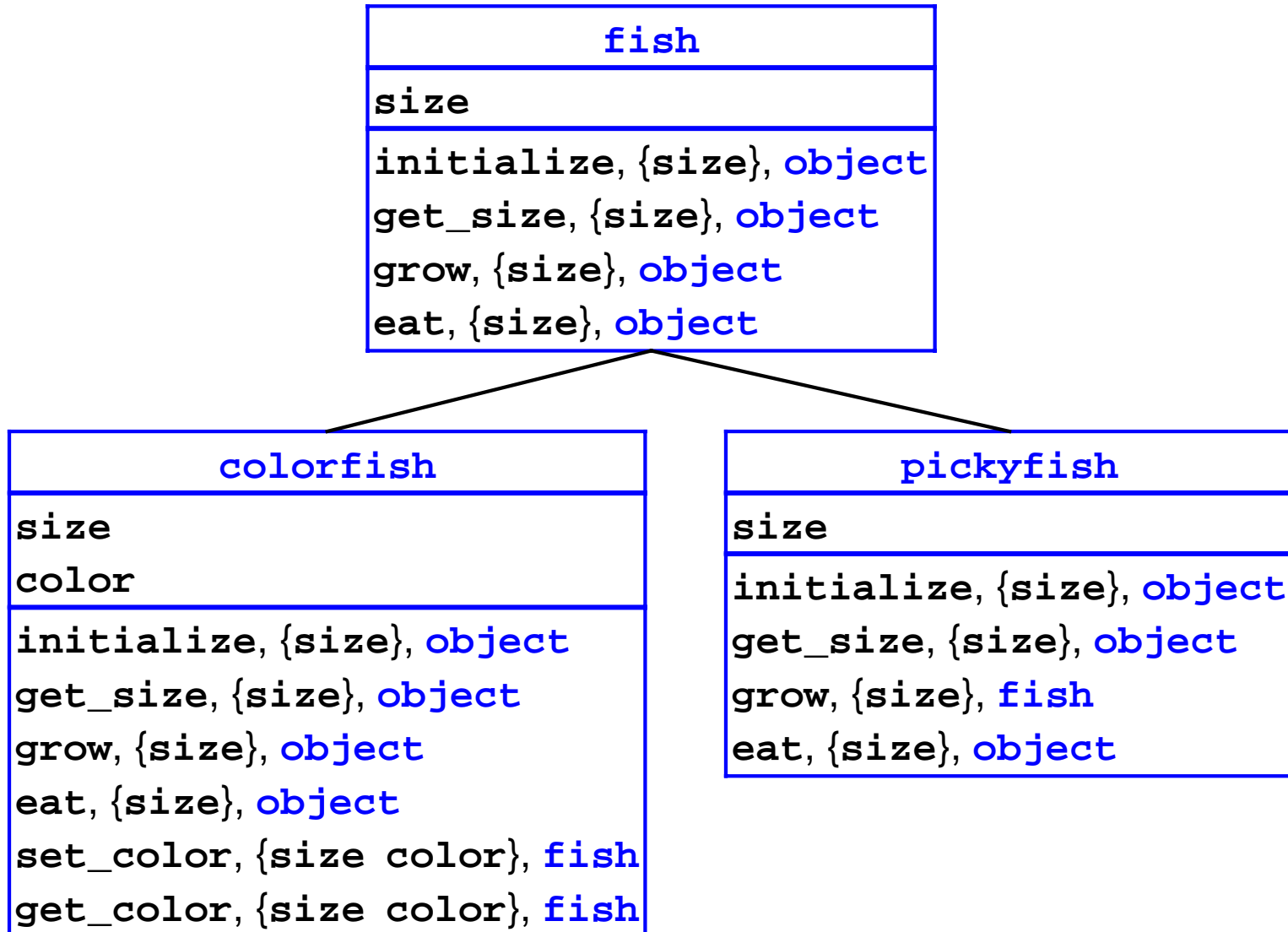
To eliminate tree walks:

2 & 3. Find method in current class, extract variable list

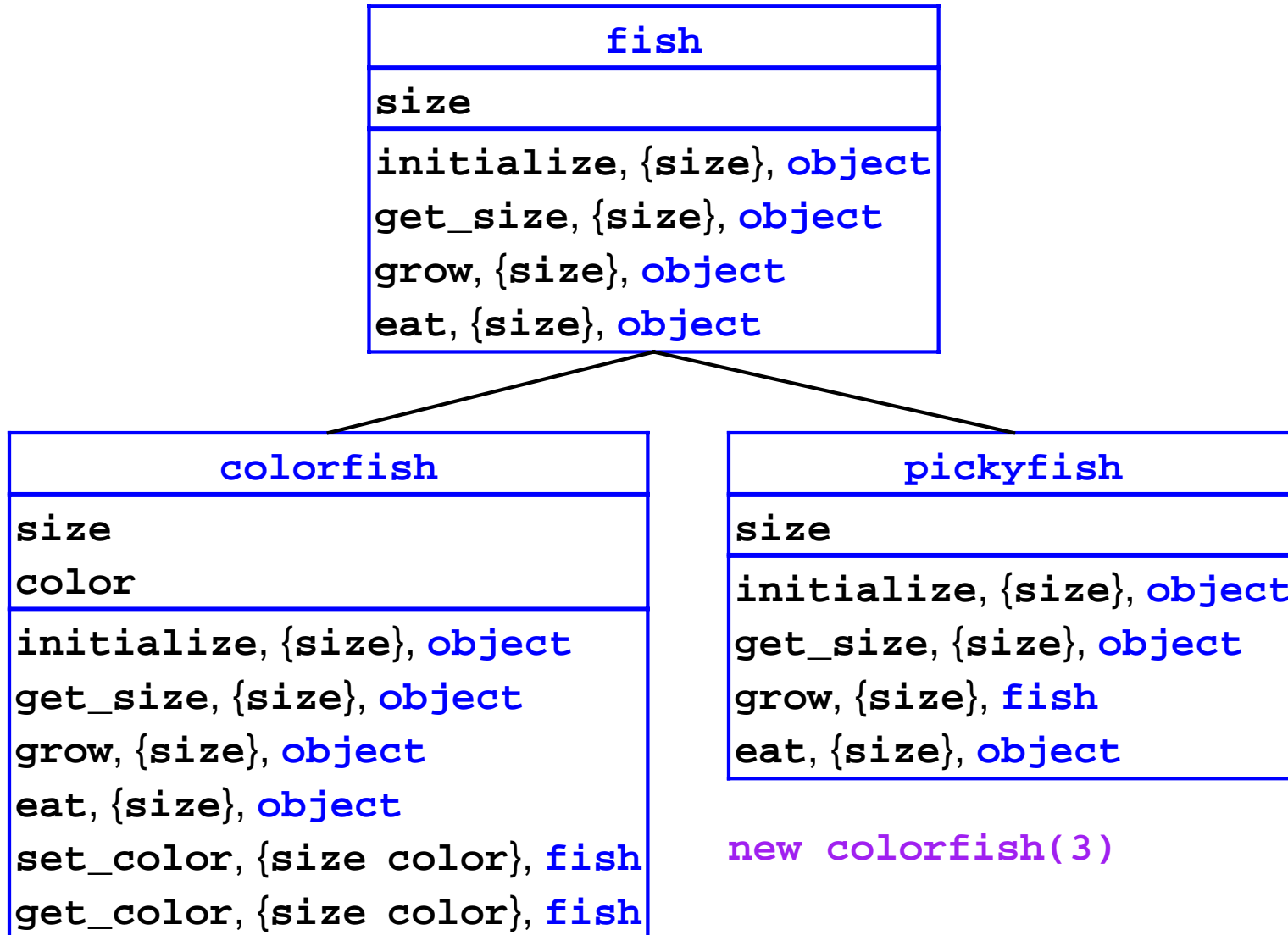
Class Elaboration



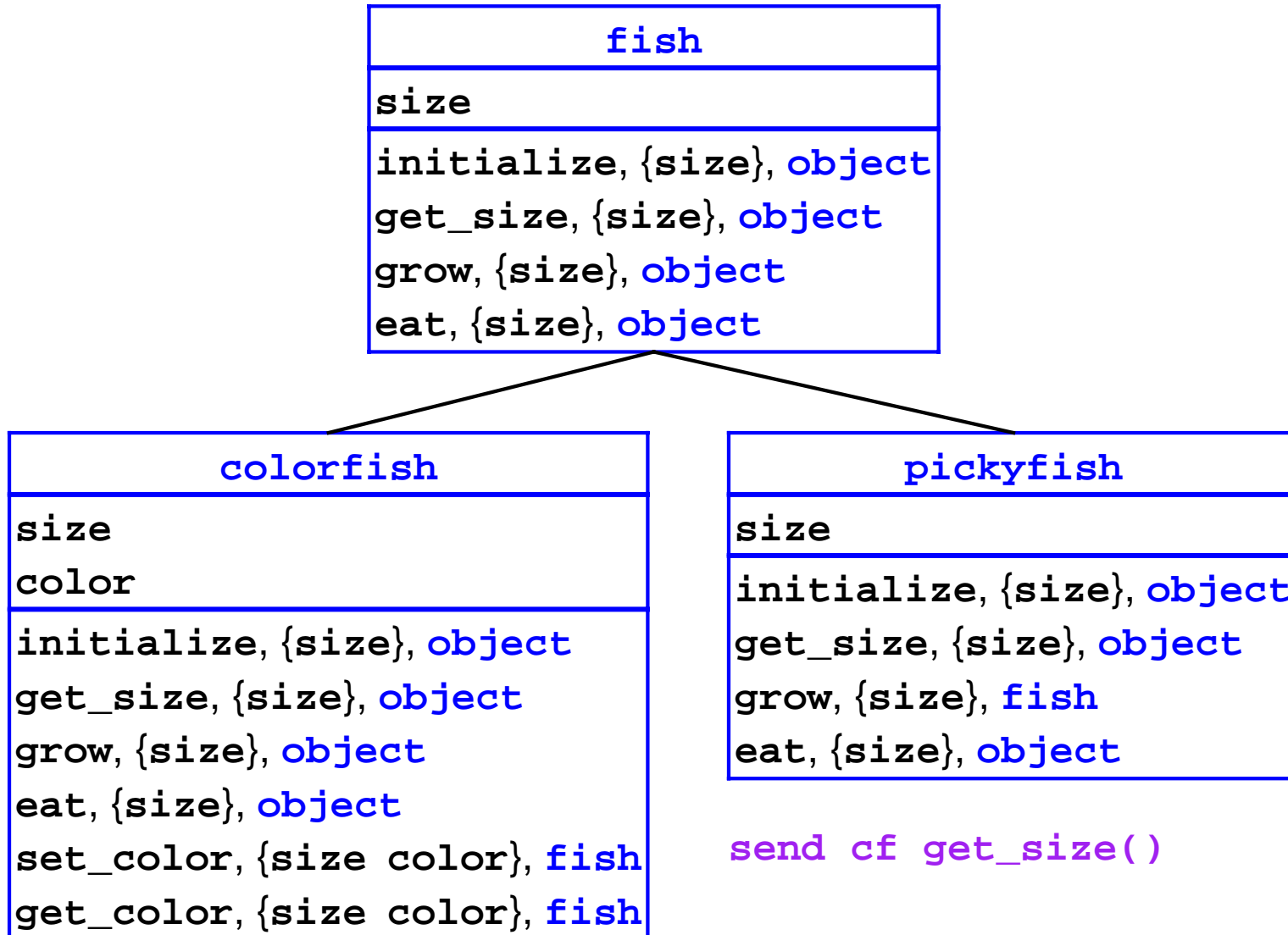
Class Elaboration



Class Elaboration



Class Elaboration



Implementation

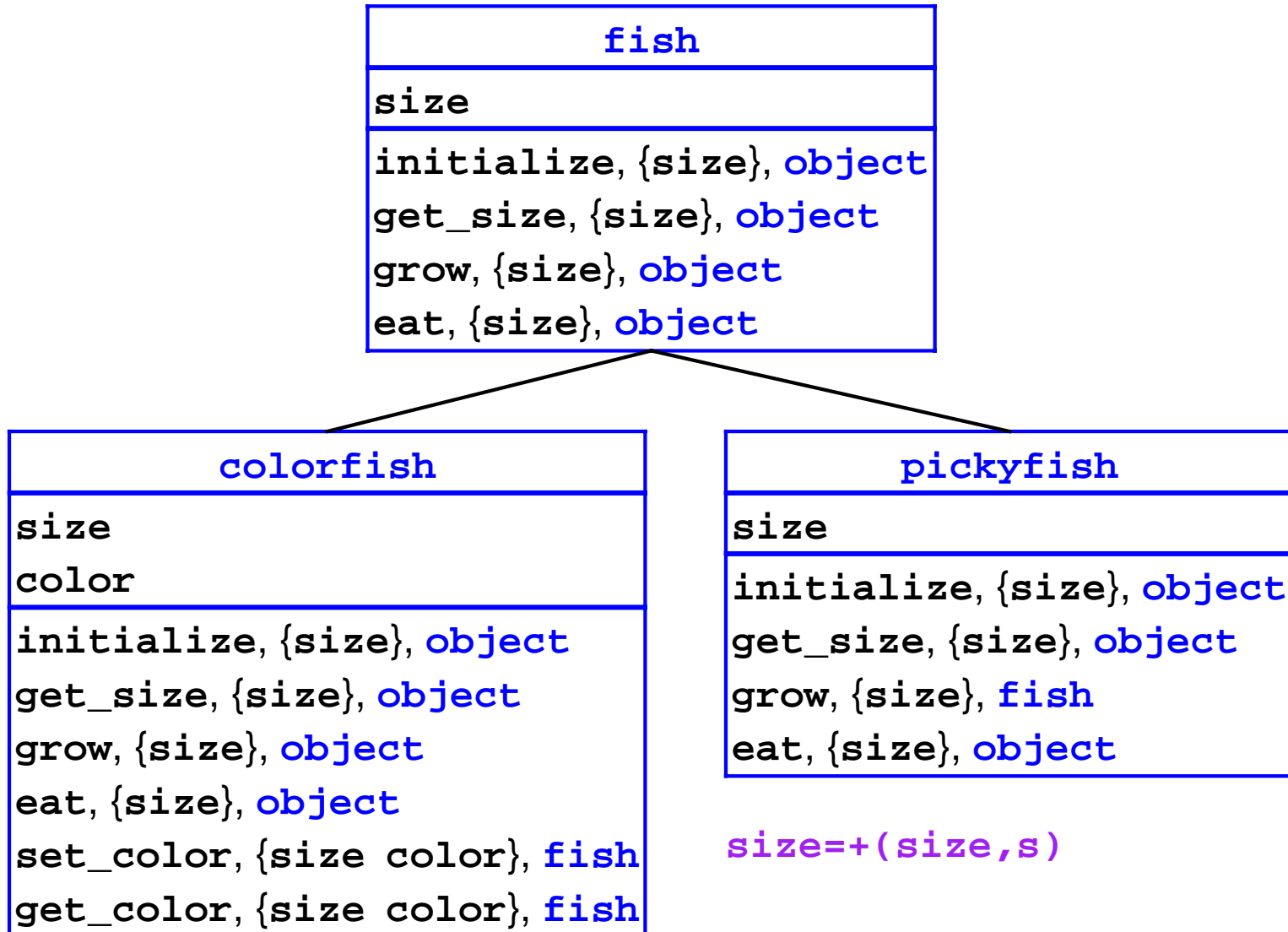
See the book and web page:

- Change `elaborate-class-decls!` to build annotated tree
- Change `new-object` to use class's immediate field list
- Change `apply-method` to work with annotated methods

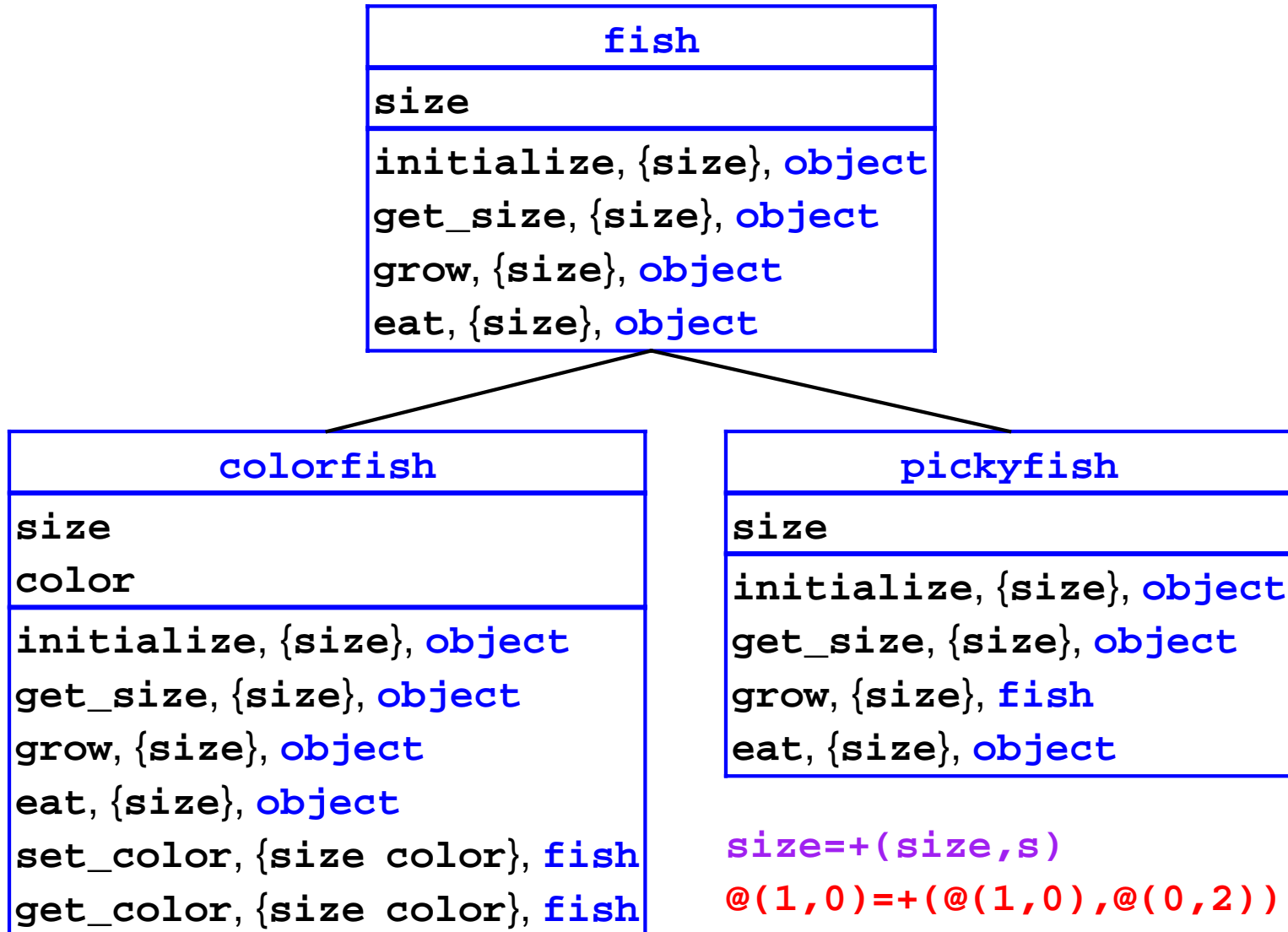
More Optimization

- Still have list walks: variable lookup, method lookup
 - Can eliminate many with lexical addresses
 - Can eliminate some by pre-computing method positions
 - Need type information to eliminate others

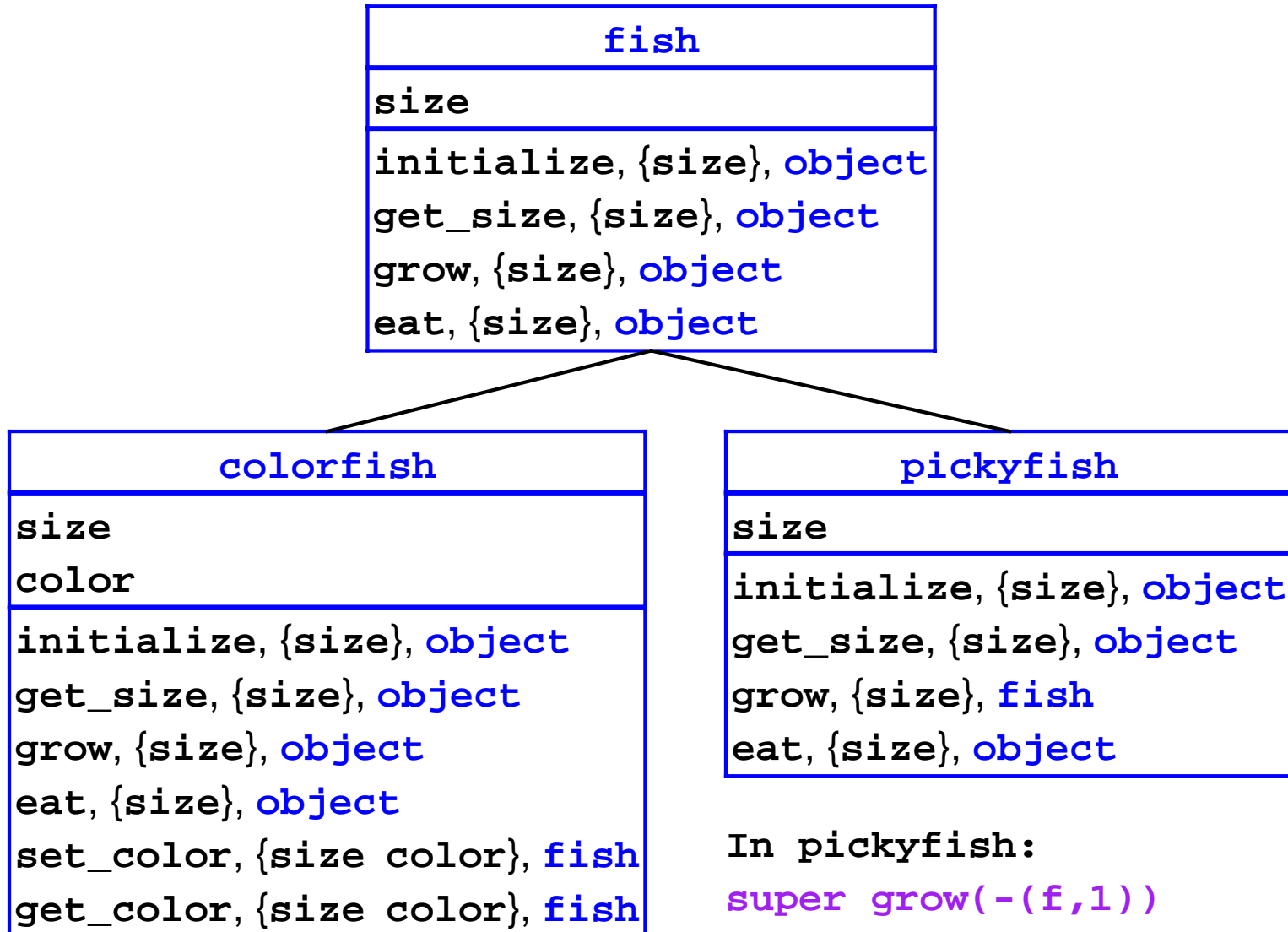
More Optimization: List Walks



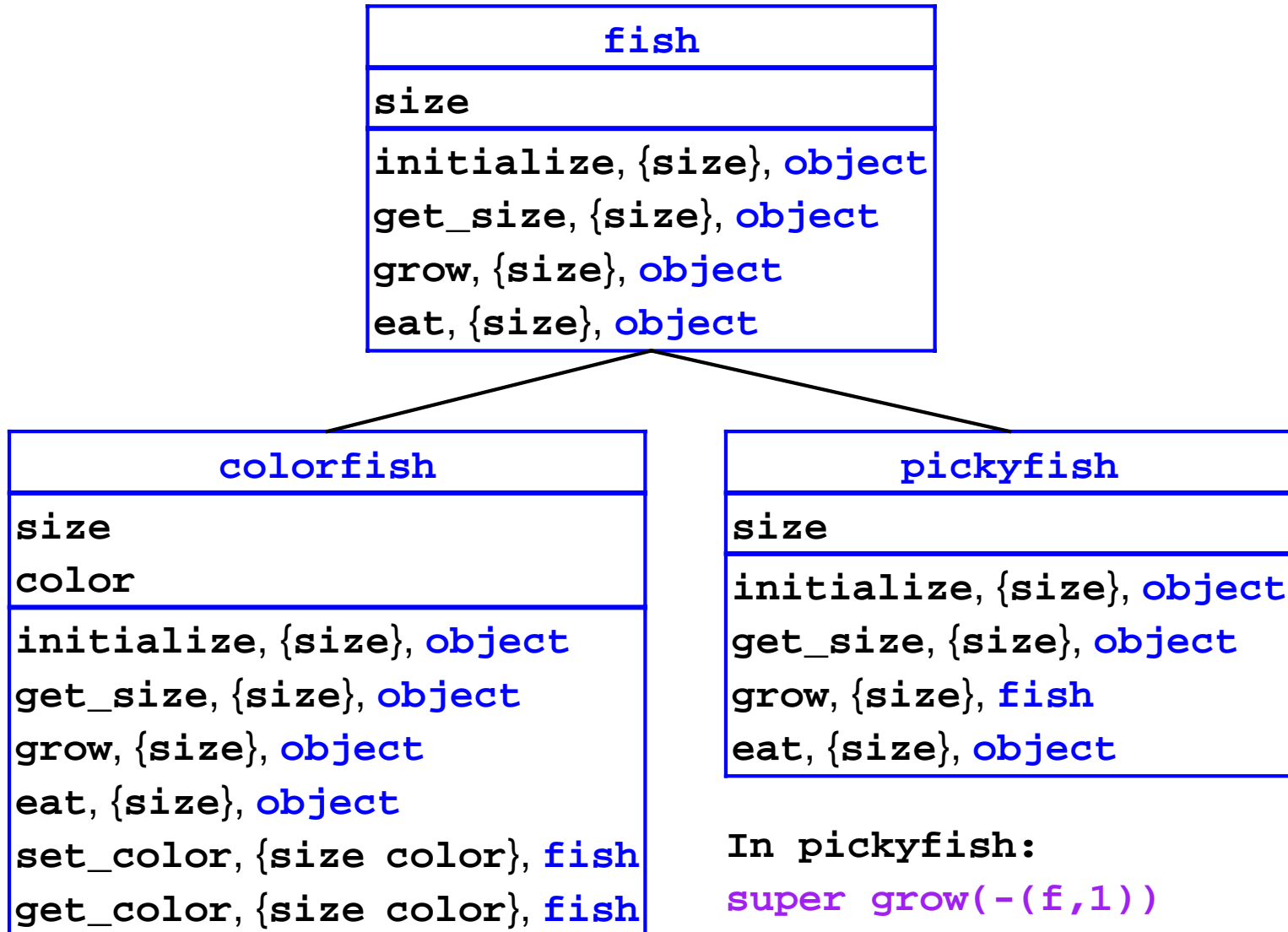
More Optimization: List Walks



More Optimization: List Walks



More Optimization: List Walks

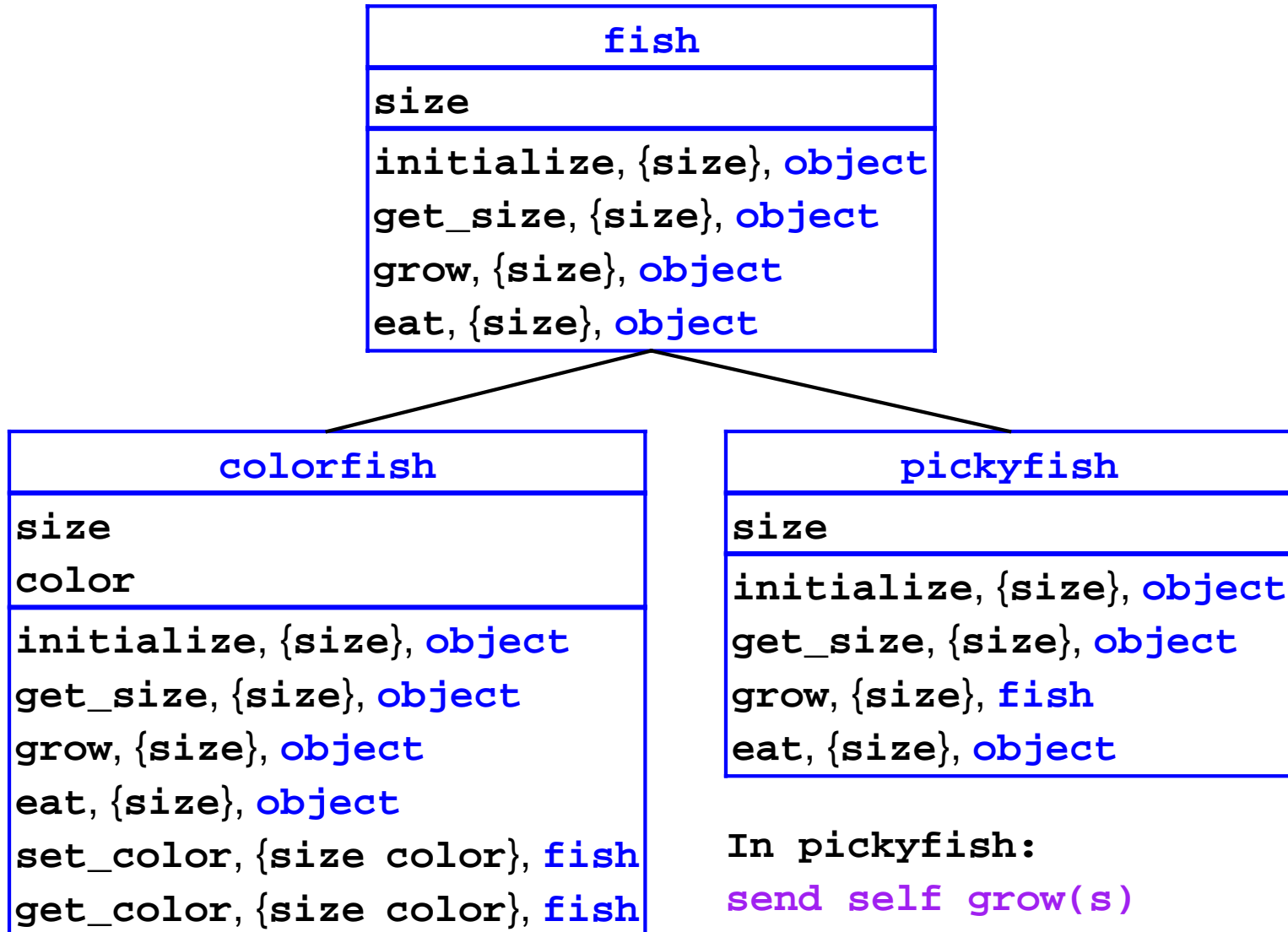


In pickyfish:

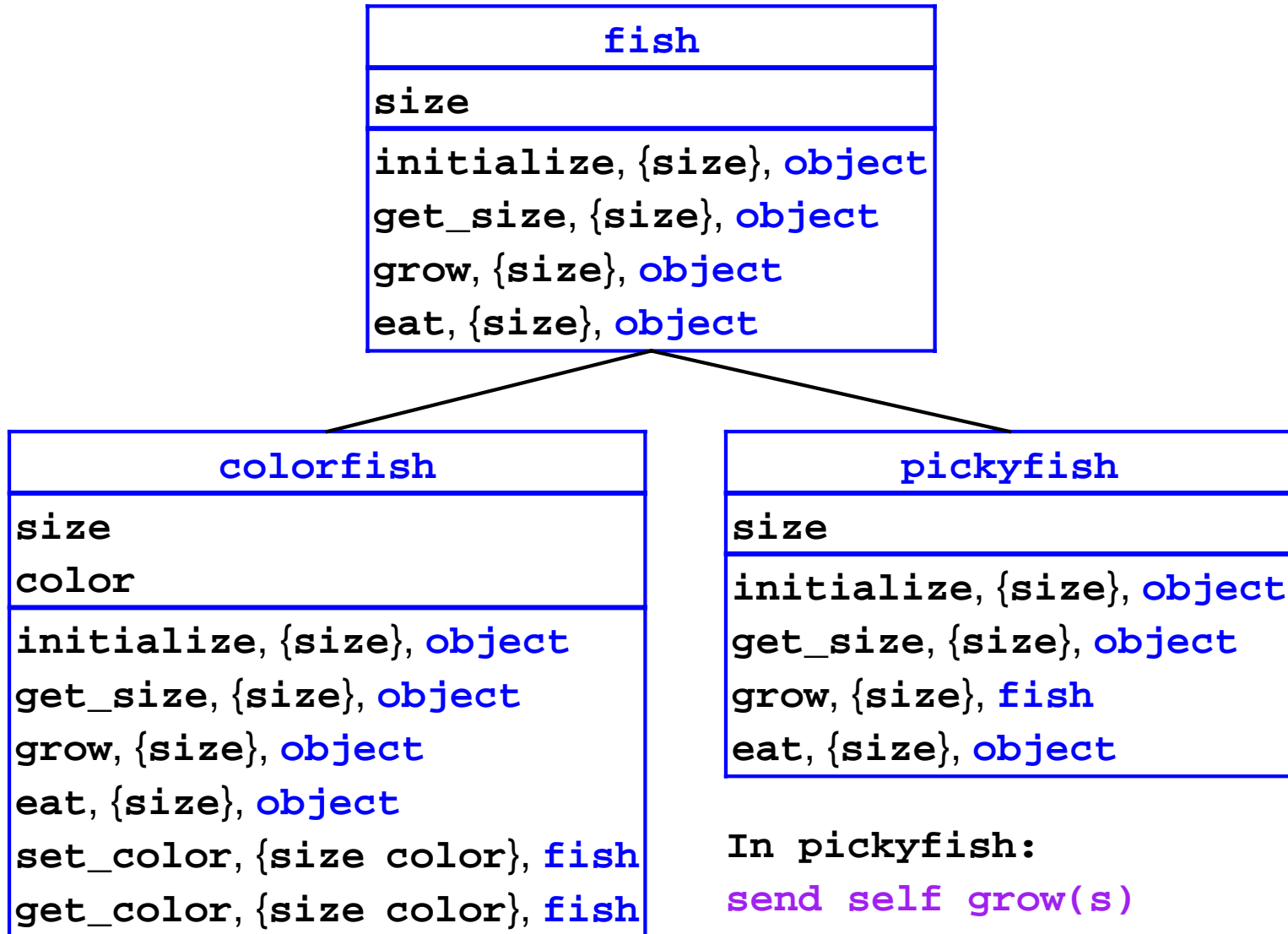
```
super grow(-(f,1))
```

```
fish.grow(-(@0,2),1))
```

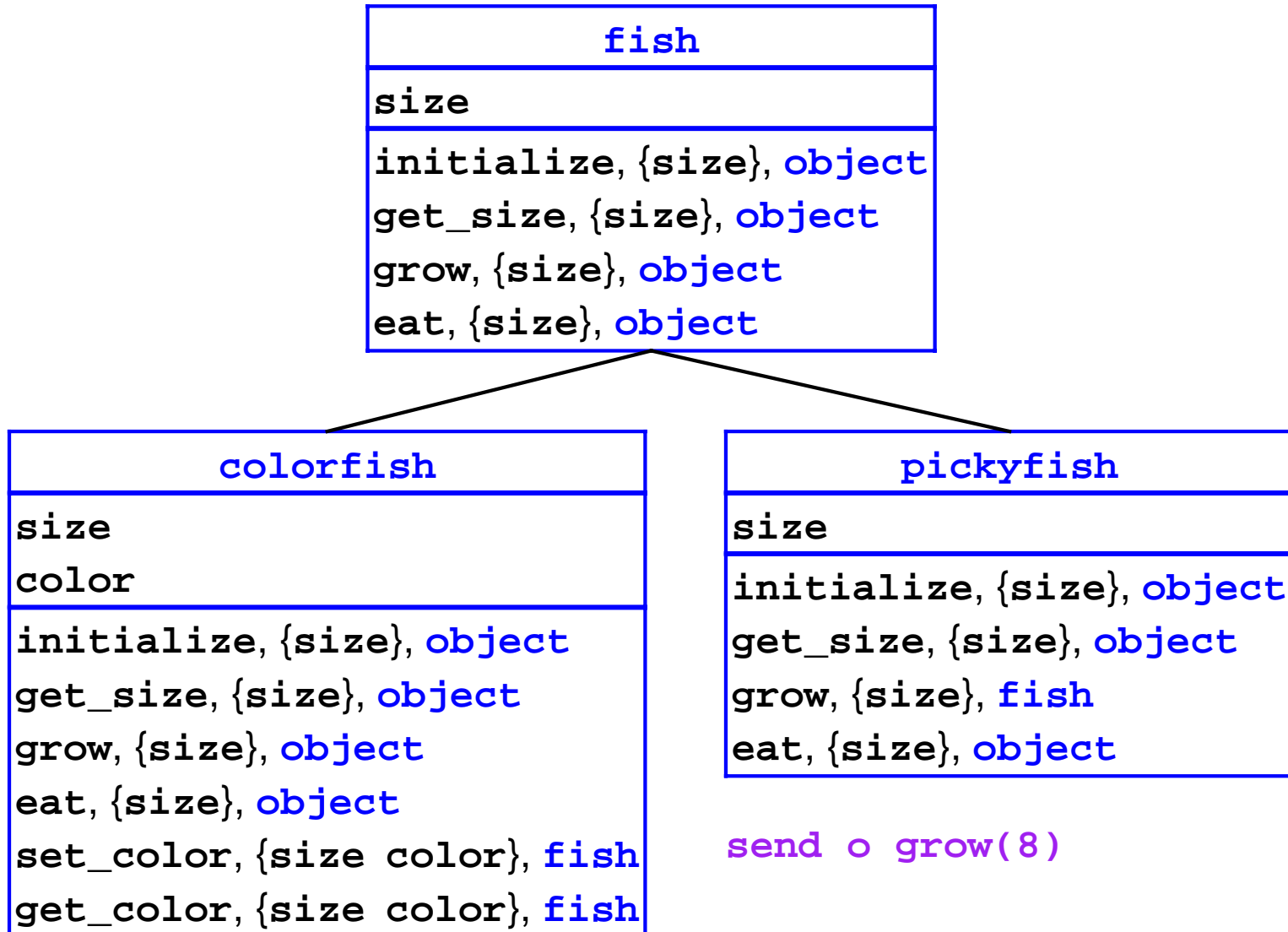
More Optimization: List Walks



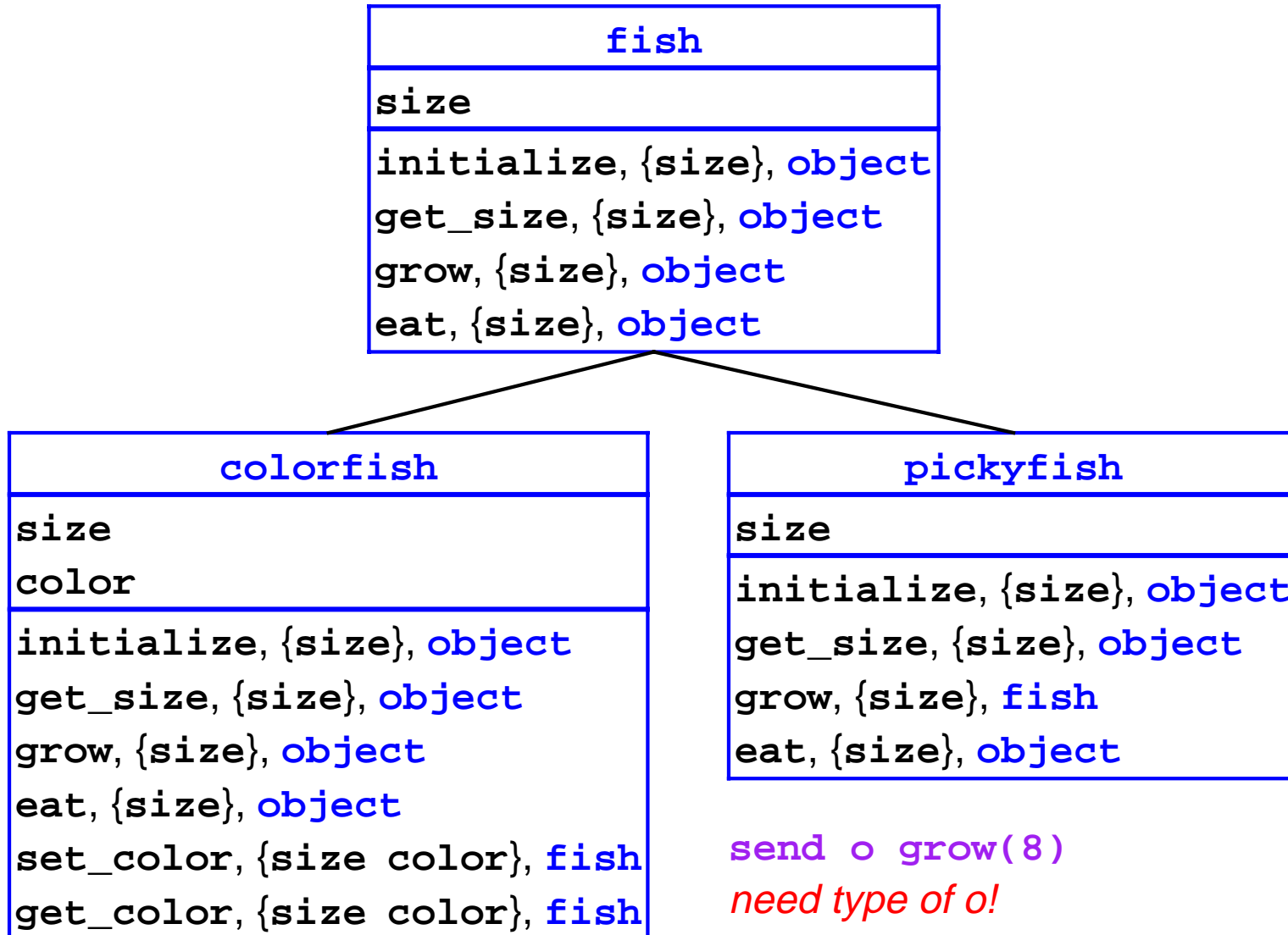
More Optimization: List Walks



More Optimization: List Walks



More Optimization: List Walks



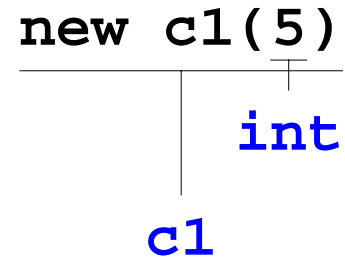
Object Types

```
new c1()  
|  
c1
```

... *if* `c1` has an `initialize` method that takes no arguments

```
class c1 extends ...  
  method void initialize() ...
```

Object Types



... if `c1` has an `initialize` method that takes one integer

```
class c1 extends ...  
  method void initialize(int v) ...
```


Object Types

```
class fish extends object
  field int size
  method void initialize (int s) ...
  method void eat(fish other) ...
class colorfish extends fish
...
```

```
send new fish(8) eat(new colorfish(1))
```

fish

colorfish

colorfish *doesn't match* fish

Subtyping

- **Subtype:** An instance of class C can be used as an instance of class C' if C is derived from C'

$$C <: C'$$

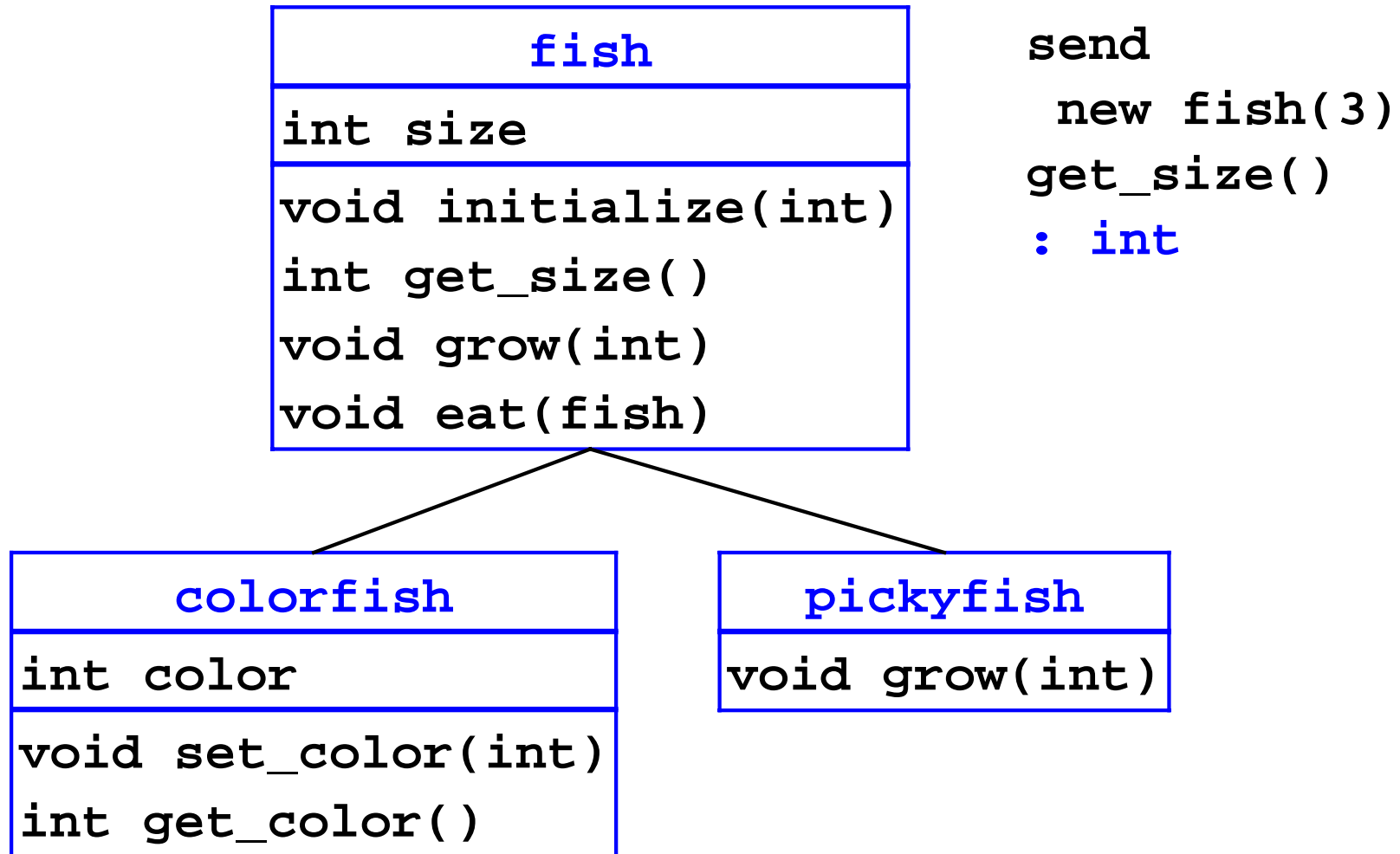
- Subtype rule:

If $e : T$ and $T <: T'$, then $e : T'$

Language Changes

- Add types to field declarations
- Add types to method arguments and result
- Add `abstract class` and `abstractmethod`
- Add `instanceof`
- Add `cast`

Program Checking



Things to Check

`cast` and `instanceof`:

- Operand has an object type (for any class)
- Target class exists

`cast o c1`

`instanceof o c7`

Things to Check

cast and **instanceof**:

- Operand has an object type (for any class)
- Target class exists

cast only:

- Class for operand and target are comparable
 - Otherwise, cast cannot possibly succeed

```
class c1 extends object ...  
class c2 extends object ...  
cast new c1() c2
```

Things to Check

Object creation:

- Class exists, and is not abstract
- Class has an `initialize` method
- `initialize`'s argument types match the operand types

```
class c1 extends object
  method void initialize(int x, bool y)
  ...

new c1(1, false)
```

Things to Check

Method calls:

- Receiver expression is an object
- Method is in the object-type's class
 - Except `initialize...`
- Method's argument types match the operand types

```
class c1 extends object
  method void initialize() ...
  method void m(int x, bool y)
  ...
let o1 = new c1()
in send o1 m(1, false)
```

Things to Check

super calls:

- Expression is within a method
- Method is in the superclass, and not abstract
- Method's argument types match the operand types

```
class c1 extends object
  method void m(int x, bool y)
```

...

```
class c2 extends c1
  method void n()
    super m(1, false)
```

...

Things to Check

`class` declarations:

- Superclass exists, and no cyclic inheritance
- Methods bodies ok
 - Use host class for type of `self`
- Overriding method signatures the same as in superclass
 - Except for `initialize`

```
class c2 extends c1
  method void m(int x, bool y)
    if y then +(2, x) else send self w()
```


The Initialize Method

```
class c1 extends obj
  field int x
  method void initialize()
    set x = 3
  method int m()
    send self initialize()
```

```
class c2 extends c1
  field int y
  method void initialize(int v)
    set y = v
    super initialize()
  ...
```

- Derived class needs different signature for `initialize`

The Initialize Method

```
class c1 extends obj
  field int x
  method void initialize()
    set x = 3
  method int m()
    send self initialize()

class c2 extends c1
  field int y
  method void initialize(int v)
    set y = v
    super initialize()
  ...
```

- Disallow `send` to `initialize`

The Initialize Method

```
class c1 extends obj
  field int x
  method void initialize()
    set x = 3
  method int m()
    send self initialize()
```

```
class c2 extends c1
  field int y
  method void initialize(int v)
    set y = v
    super initialize()
  ...
```

- super call to initialize is ok

Field Initializations

Not checked: field initializations

```
class interior_node extends tree
  field tree left
  field tree right
  method void initialize(tree l, tree r)
    begin
      send left sum();
      ...
    end
```

- Can get "bad object 0 for method call"
- This is analogous to the `null` error in Java

Type Checking and Errors

Disallowed errors:

- Object has no such method, or Super method not found
- Can't call method of non-object, non-0
- No such field, no such variable
- Illegal primitive argument (except car of empty)

Allowed errors:

- Can't call method of 0
- Cast failed
- Car of empty

Implementation

See the book and web page

Mixing Subtyping and Procedures

Our language still has procedures:

```
let feed = proc(colorfish f)
            send f grow(10)
    o1 = new colorfish(0)
in
    (feed o1)
```

Mixing Subtyping and Procedures

And higher-order procedures:

```
let feed = proc(colorfish f)
            send f grow(10)
    o1 = new colorfish(0)
    o2 = new colorfish(1)
in let toboth = proc((colorfish -> void) p)
    begin
        (p o1);
        (p o2)
    end
    in (toboth feed)
```


Mixing Subtyping and Procedures

Subtyping on procedure arguments:

```
let feed = proc(fish f)  
             send f grow(10)  
    o1 = new colorfish(0)  
in  
    (feed o1)
```

- This works, and is allowed by our subtyping rule

Mixing Subtyping and Procedures

Subtyping on procedure arguments:

```
let feed = proc(fish f)
    send f grow(10)
    o1 = new colorfish(0)
    o2 = new colorfish(1)
in let toboth = proc((colorfish -> void) p)
    begin
        (p o1);
        (p o2)
    end
    in (toboth feed)
```

- This works, but is *not* allowed by our subtyping rule

(**fish** -> void) versus (colorfish -> void)

Procedure Subtyping Rule

If $T1 <: T1'$ and $T2 <: T2'$

then $(T1' \rightarrow T2) <: (T1 \rightarrow T2')$

Another example:

- **dog <: animal**
 - a dog can go anywhere an animal can go
- **(animal -> hairstyle) <: (dog -> hairstyle)**
 - a groomer for all animals can groom a dog
 - a groomer who only works with dogs doesn't work for all animals

Procedure Subtyping Rule

If $T1 <: T1'$ and $T2 <: T2'$

then $(T1' \rightarrow T2) <: (T1 \rightarrow T2')$

General intuition:

- $T1 <: T1'$ means $T1'$ is more general than $T1$



- A function that is willing to accept a more general argument is itself more specific



Procedure Subtyping Rule

If $T1 <: T1'$ and $T2 <: T2'$

then $(T1' \rightarrow T2) <: (T1 \rightarrow T2')$

- Procedure types are ***contravariant*** with respect to their argument types
- Procedure types are ***covariant*** with respect to their result types