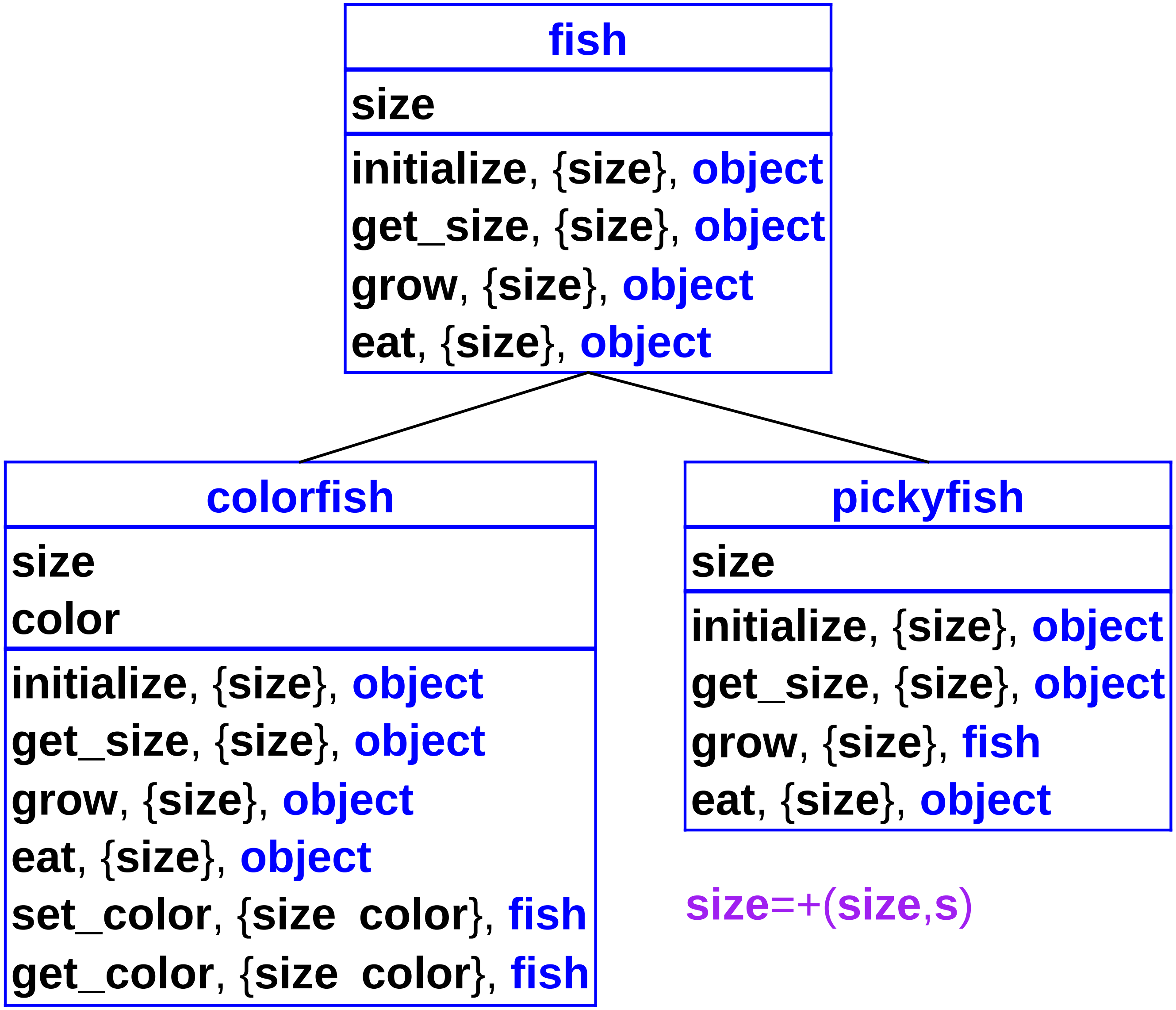
More Optimization

- 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: Eliminating List Walks



fish

size

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object

colorfish

size
color

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object
set_color, {size color}, fish
get_color, {size color}, fish

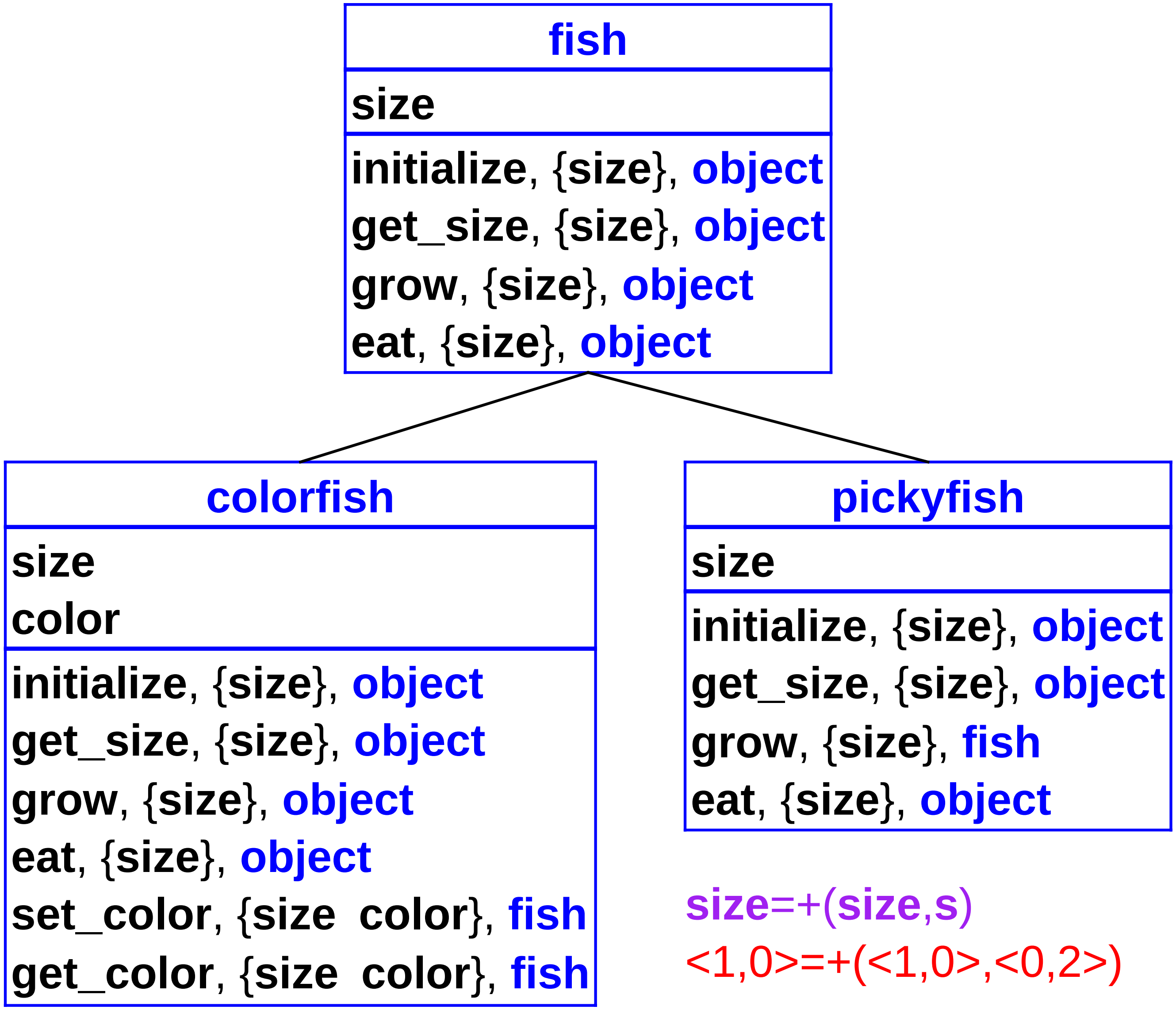
pickyfish

size

initialize, {size}, object
get_size, {size}, object
grow, {size}, fish
eat, {size}, object

size=+(size,s)

More Optimization: Eliminating List Walks



fish

size

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object

colorfish

size
color

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object
set_color, {size color}, fish
get_color, {size color}, fish

pickyfish

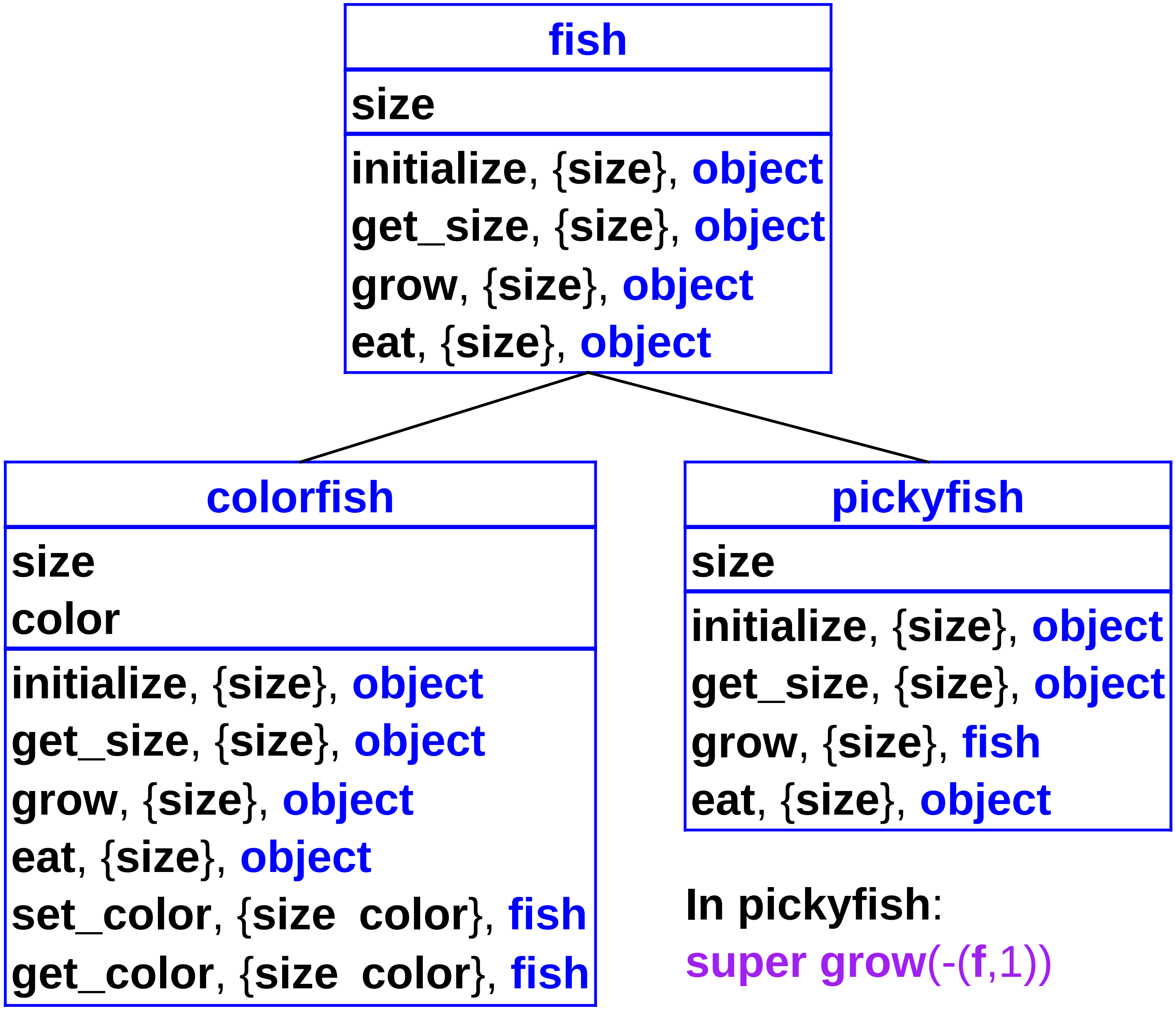
size

initialize, {size}, object
get_size, {size}, object
grow, {size}, fish
eat, {size}, object

size=+(size,s)

<1,0>=+(<1,0>,<0,2>)

More Optimization: Eliminating List Walks



fish

size

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **object**
eat, {**size**}, **object**

colorfish

size
color

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **object**
eat, {**size**}, **object**
set_color, {**size color**}, **fish**
get_color, {**size color**}, **fish**

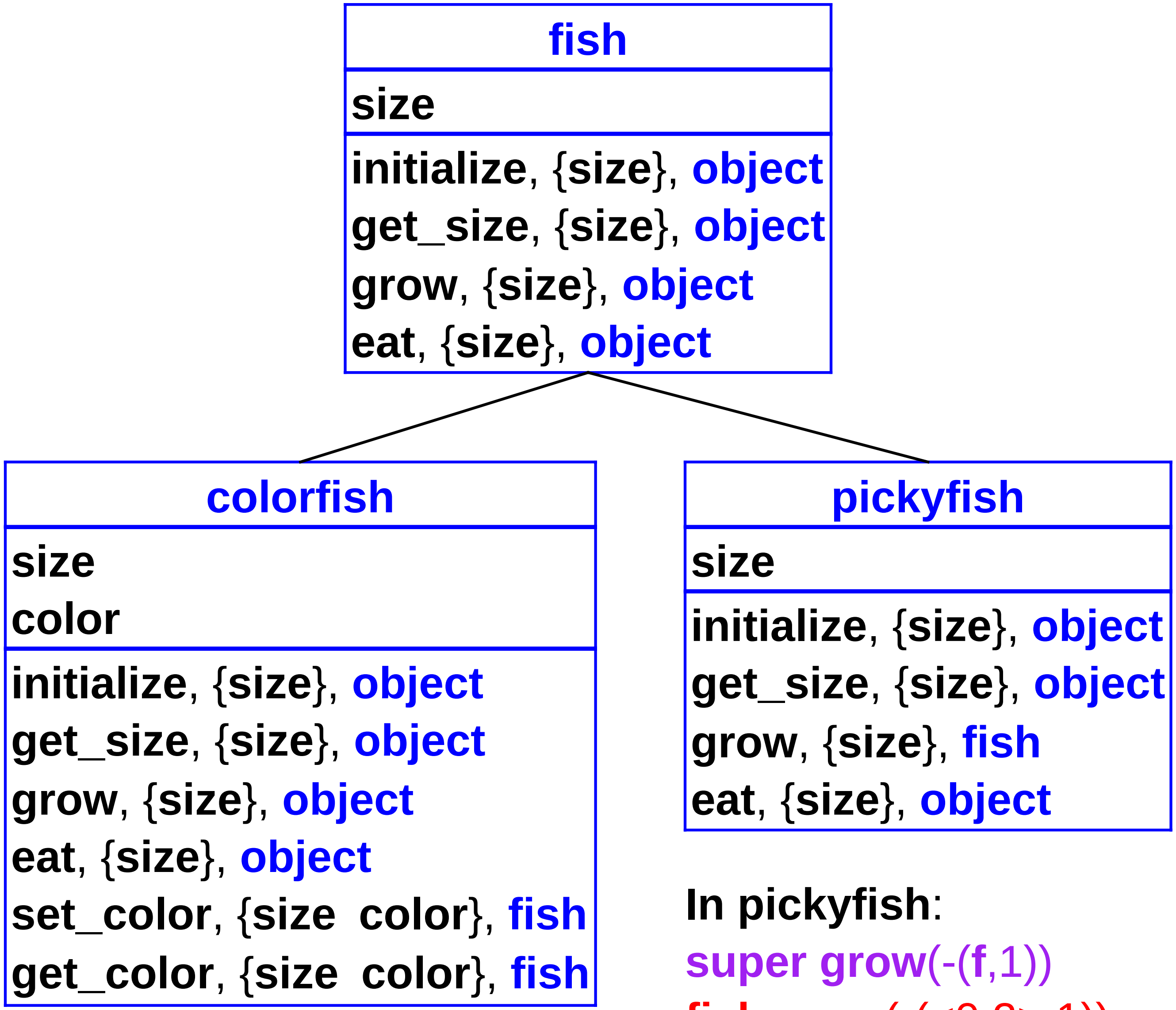
pickyfish

size

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **fish**
eat, {**size**}, **object**

In pickyfish:
super grow(-(**f**,1))

More Optimization: Eliminating List Walks



fish

size

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **object**
eat, {**size**}, **object**

colorfish

size
color

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **object**
eat, {**size**}, **object**
set_color, {**size color**}, **fish**
get_color, {**size color**}, **fish**

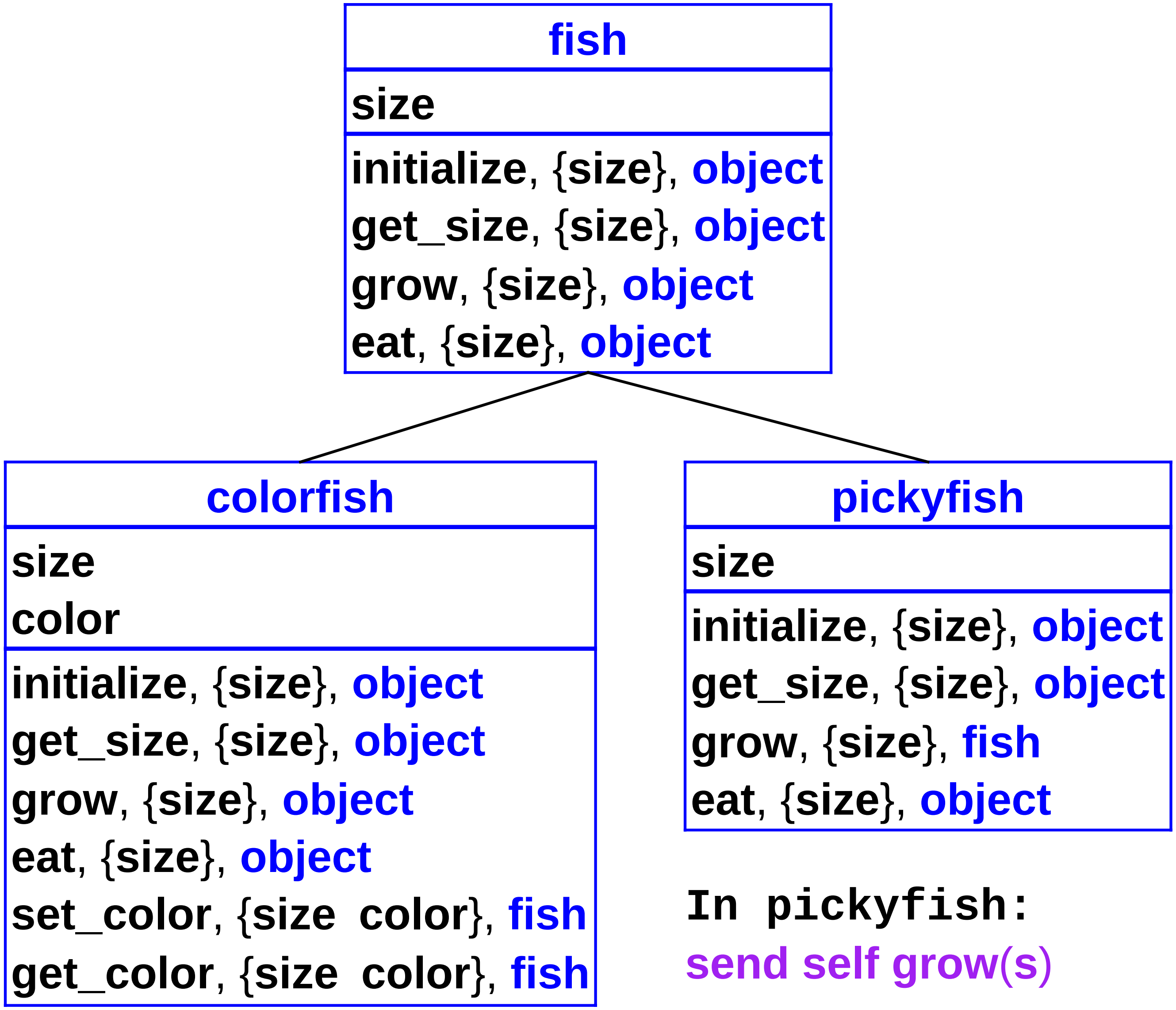
pickyfish

size

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **fish**
eat, {**size**}, **object**

In pickyfish:
super grow(-(**f**,1))
fish.grow(-(<0,2>,1))

More Optimization: Eliminating List Walks



fish

size

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object

colorfish

size
color

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object
set_color, {size color}, fish
get_color, {size color}, fish

pickyfish

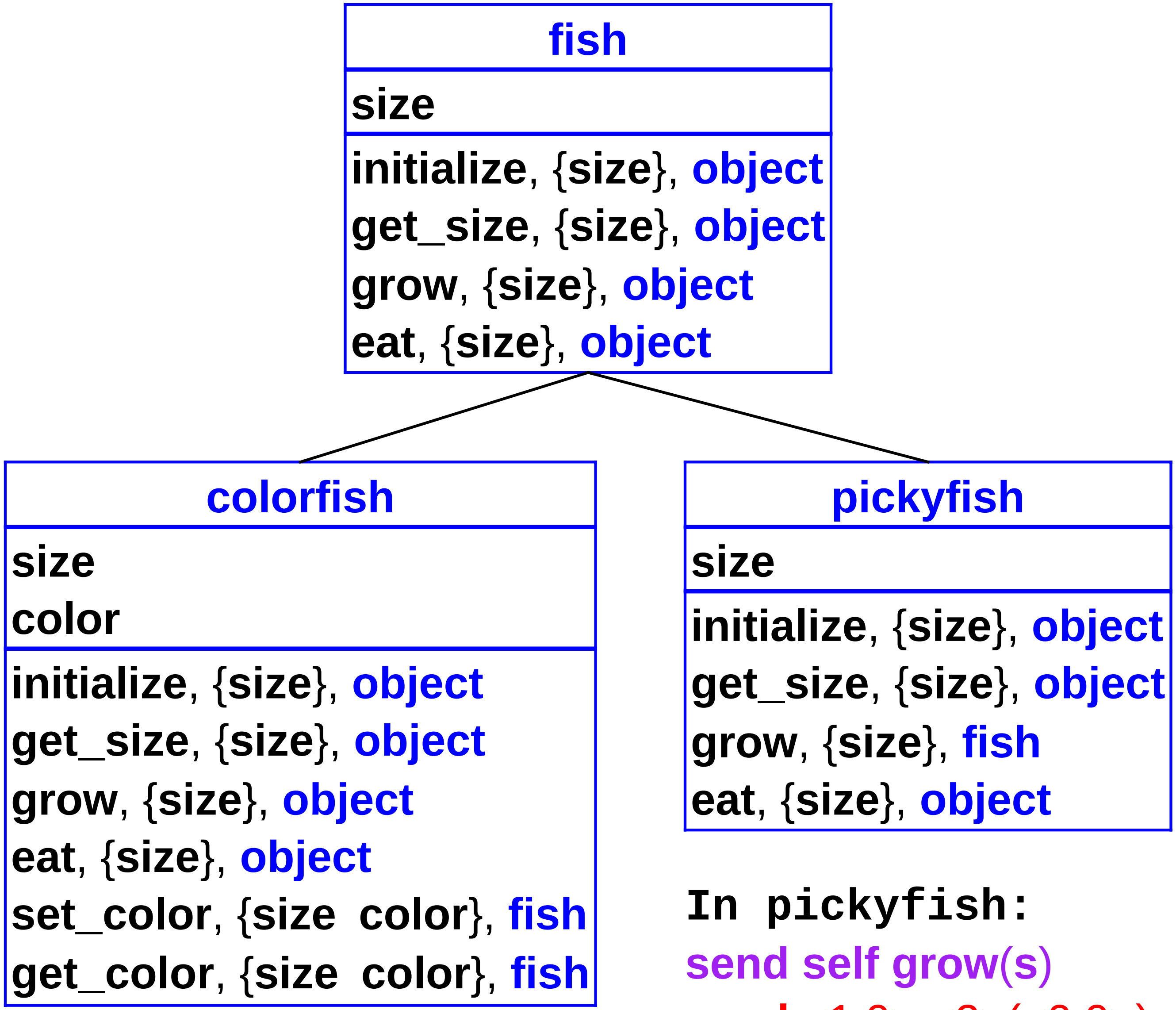
size

initialize, {size}, object
get_size, {size}, object
grow, {size}, fish
eat, {size}, object

`In pickyfish:`

send self grow(s)

More Optimization: Eliminating List Walks



fish

size

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object

colorfish

size
color

initialize, {size}, object
get_size, {size}, object
grow, {size}, object
eat, {size}, object
set_color, {size color}, fish
get_color, {size color}, fish

pickyfish

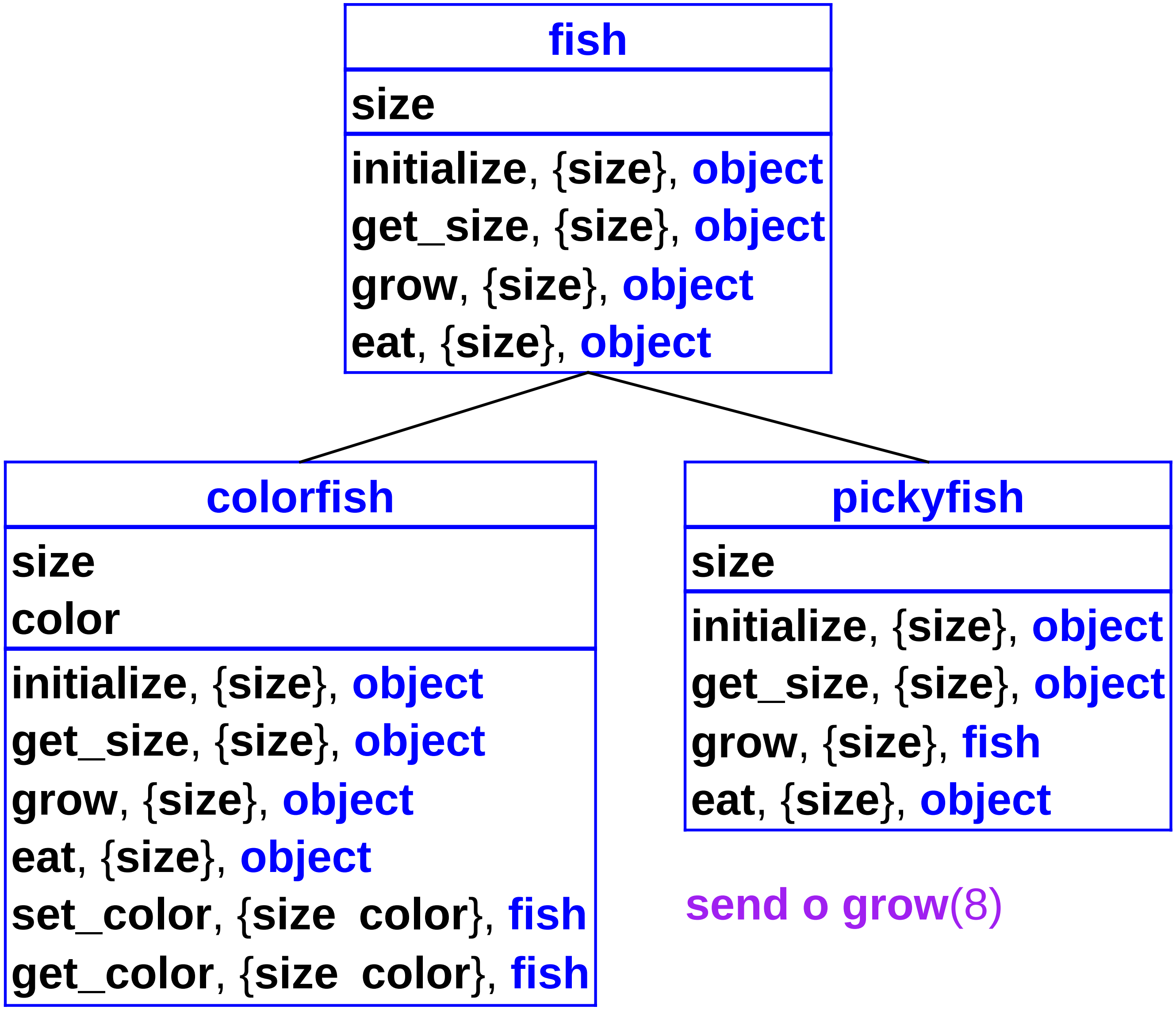
size

initialize, {size}, object
get_size, {size}, object
grow, {size}, fish
eat, {size}, object

```
In pickyfish:
```

send self grow(s)
send <1,0> <2>(<0,0>)

More Optimization: Eliminating List Walks



fish

size

initialize, {size}, **object**
get_size, {size}, **object**
grow, {size}, **object**
eat, {size}, **object**

colorfish

size
color

initialize, {size}, **object**
get_size, {size}, **object**
grow, {size}, **object**
eat, {size}, **object**
set_color, {size color}, **fish**
get_color, {size color}, **fish**

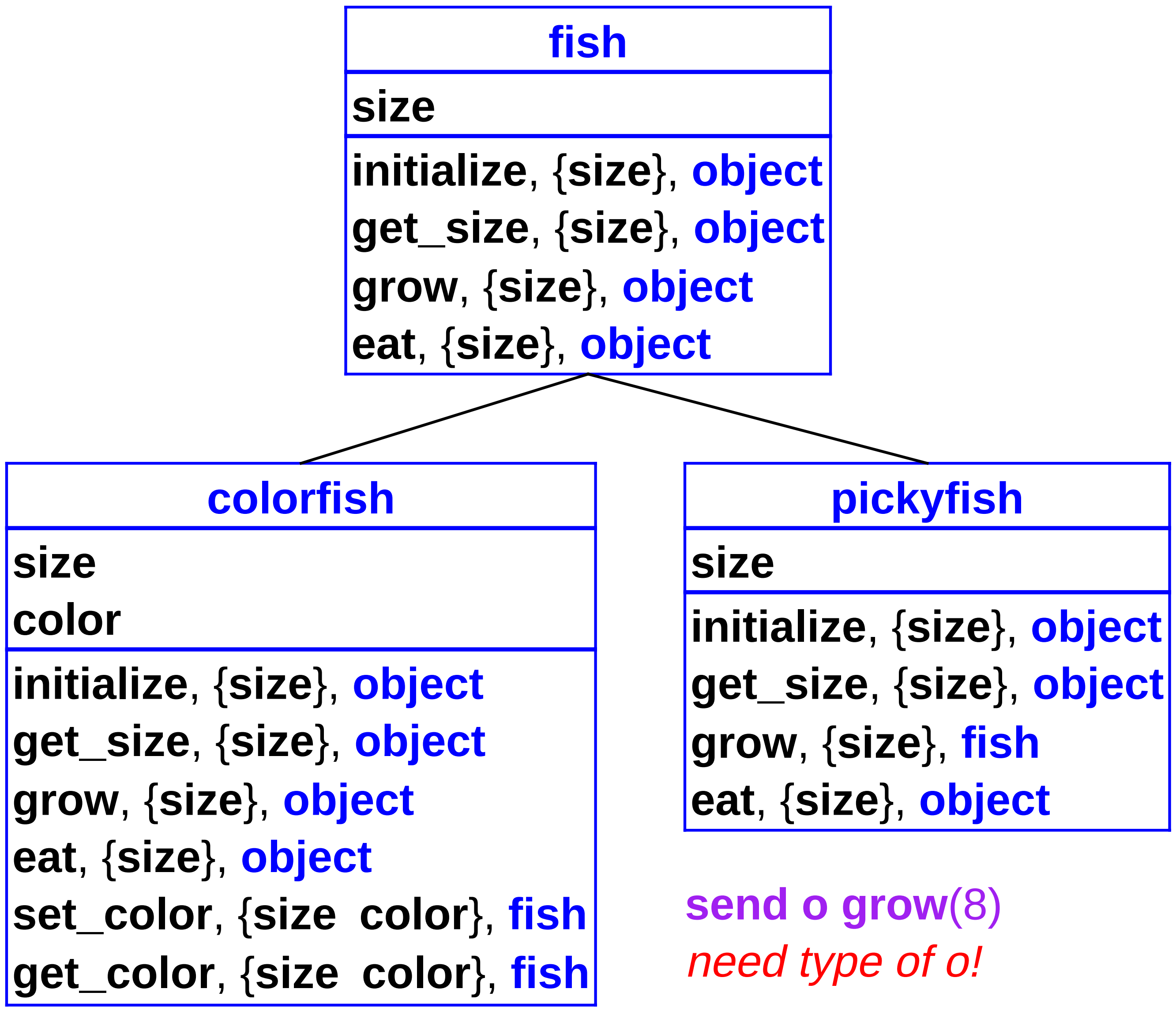
pickyfish

size

initialize, {size}, **object**
get_size, {size}, **object**
grow, {size}, **fish**
eat, {size}, **object**

send o grow(8)

More Optimization: Eliminating List Walks



fish

size

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **object**
eat, {**size**}, **object**

colorfish

size
color

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **object**
eat, {**size**}, **object**
set_color, {**size color**}, **fish**
get_color, {**size color**}, **fish**

pickyfish

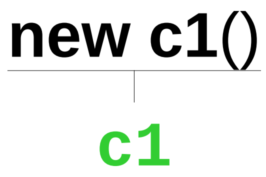
size

initialize, {**size**}, **object**
get_size, {**size**}, **object**
grow, {**size**}, **fish**
eat, {**size**}, **object**

send o grow(8)

need type of o!

Object Types



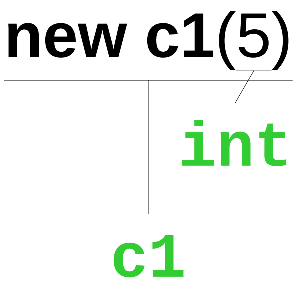
new c1()

c1

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

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

Object Types



new c1(5)

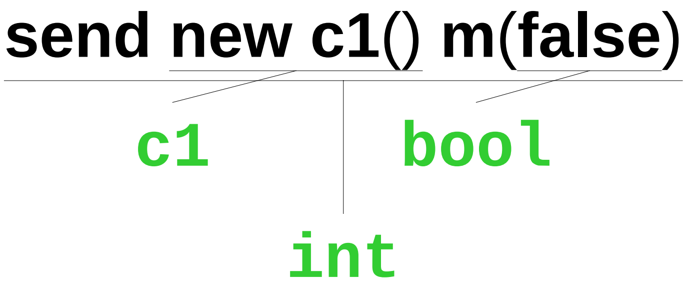
int

c1

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

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

Object Types

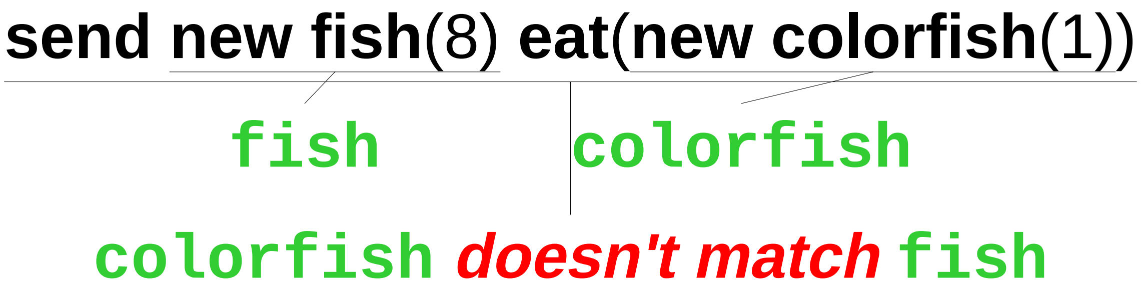


... if **c1** has an **m** method that takes **`bool`** and returns **`int`**

```
class c1 extends ...
 method void initialize() ...
 method int m(bool 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'**

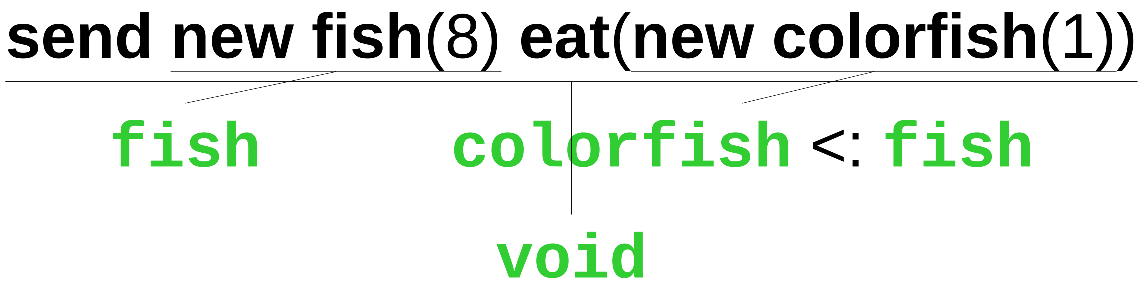
- Subtype rule:

If $E \vdash e : T_1$ and $T_1 <: T_2$, then $E \vdash e : T_2$

$$\frac{E \vdash e : T_1 \qquad\qquad T_1 <: T_2}{E \vdash e : T_2}$$

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 <: fish

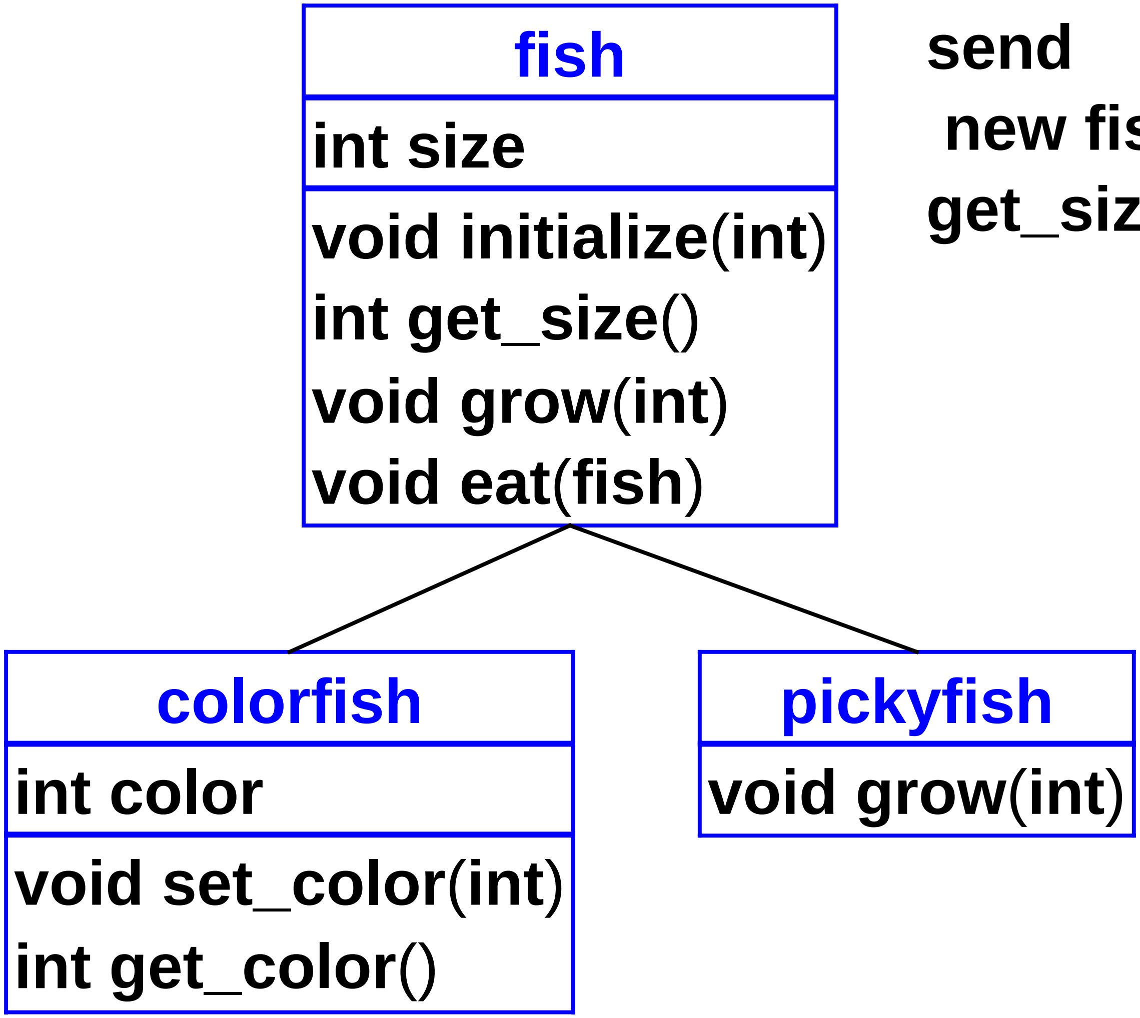
void

Language Changes

- Add types to field declarations
- Add types to method arguments and result

- Add **abstract class** and **abstractmethod**
- Add **cast**

Program Checking



fish

int size

void initialize(int)
int get_size()
void grow(int)
void eat(fish)

colorfish

int color

void set_color(int)
int get_color()

pickyfish

void grow(int)

```
send
 new fish(3)
get_size()   : int
```

Things to Check

cast:

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

cast o c1

Things to Check

cast:

- Operand has an object type (for any class)
- Target class exists
- Class for operand and target must be 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 are the same as in superclass
    - Except for **initialize**

```
class c2 extends c1
  method int 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

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

$(\texttt{fish} \rightarrow \texttt{void})$ versus $(\texttt{colorfish} \rightarrow \texttt{void})$

Procedure Subtyping Rule

If T_1 <: T_{10} and T_2 <: T_{20}

then $(T_{10} \rightarrow T_2)$ <: $(T_1 \rightarrow T_{20})$

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

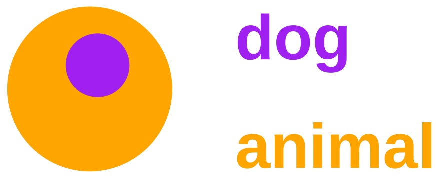
Procedure Subtyping Rule

If $T_1 <: T_{10}$ and $T_2 <: T_{20}$

then $(T_{10} \rightarrow T_2) <: (T_1 \rightarrow T_{20})$

General intuition:

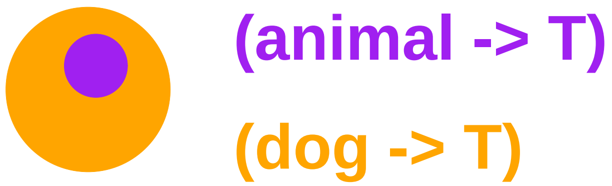
- $T_1 <: T_{10}$ means T_{10} is more general than T_1



dog

animal

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



(animal -> T)

(dog -> T)

Procedure Subtyping Rule

If T_1 <: T_{10} and T_2 <: T_{20}

then $(T_{10} \rightarrow T_2)$ <: $(T_1 \rightarrow T_{20})$

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