# Perceus
## Garbage Free Reference Counting with Reuse

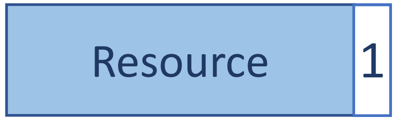
Ningning Xie

香 港 大 學
THE UNIVERSITY OF HONG KONG

Joint work with Alex Reinking,
Leonardo de Moura, and Daan Leijen

Microsoft Research

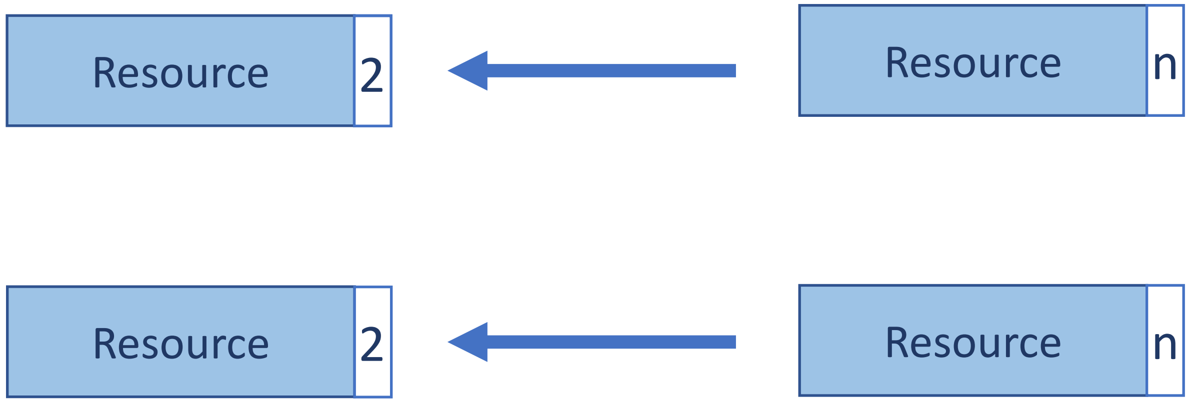
# Reference Counting 101



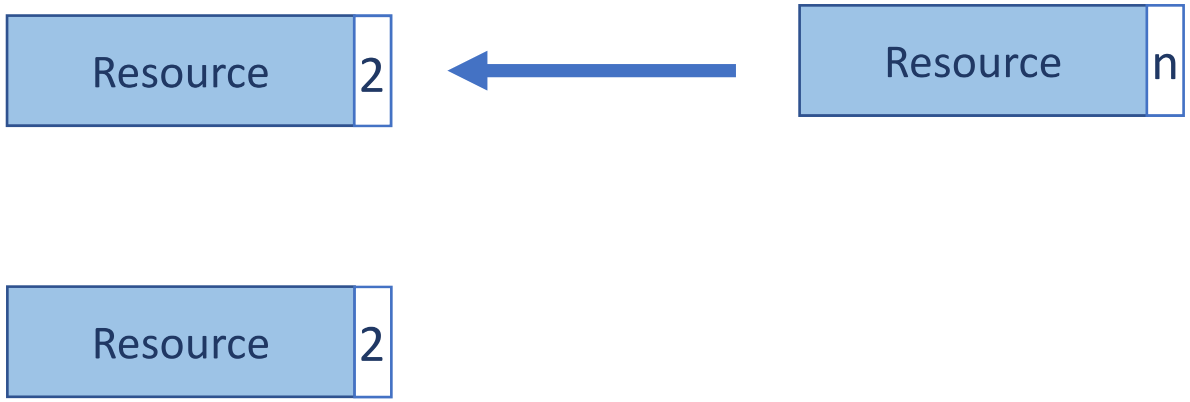
# Reference Counting 101

# Reference Counting 101

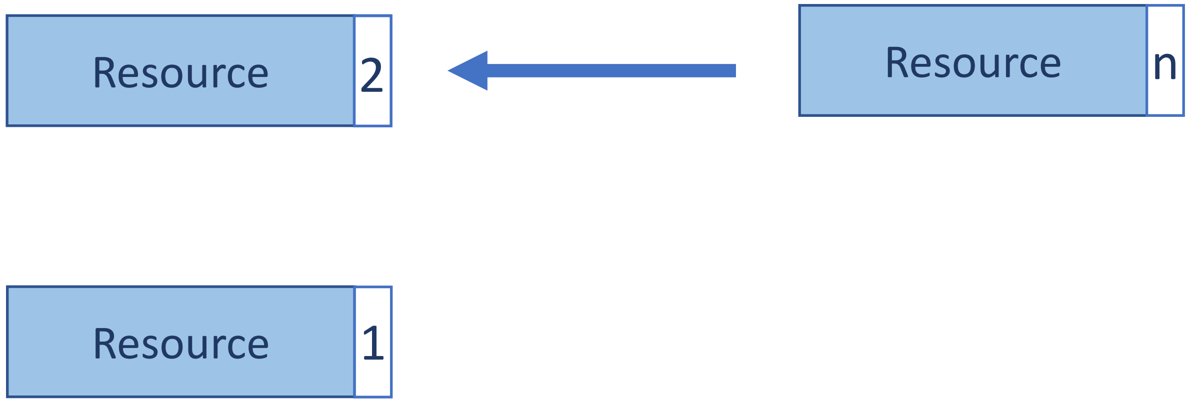
# Reference Counting 101



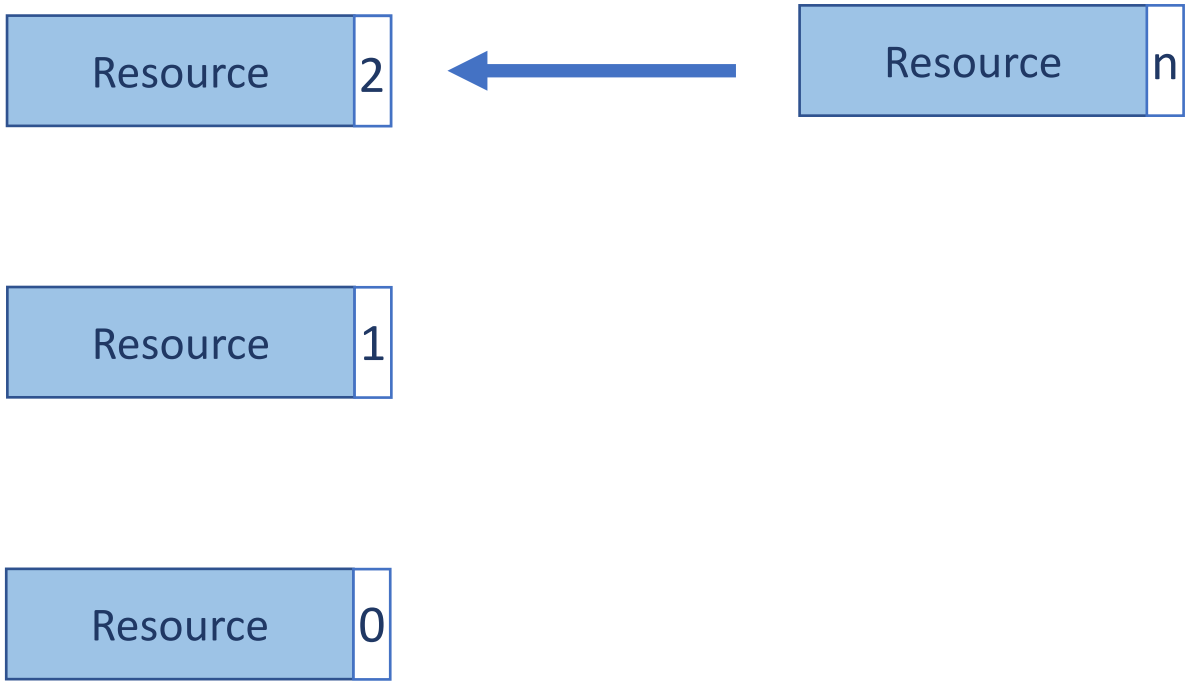
# Reference Counting 101



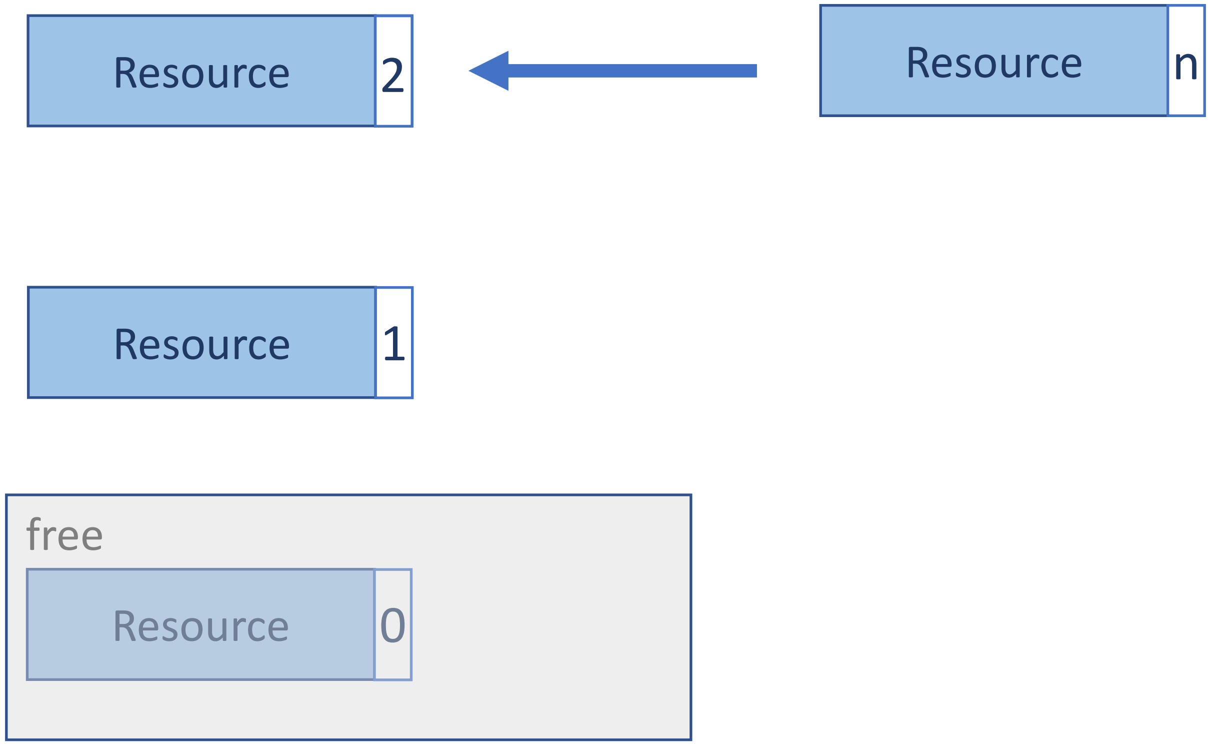
# Reference Counting 101



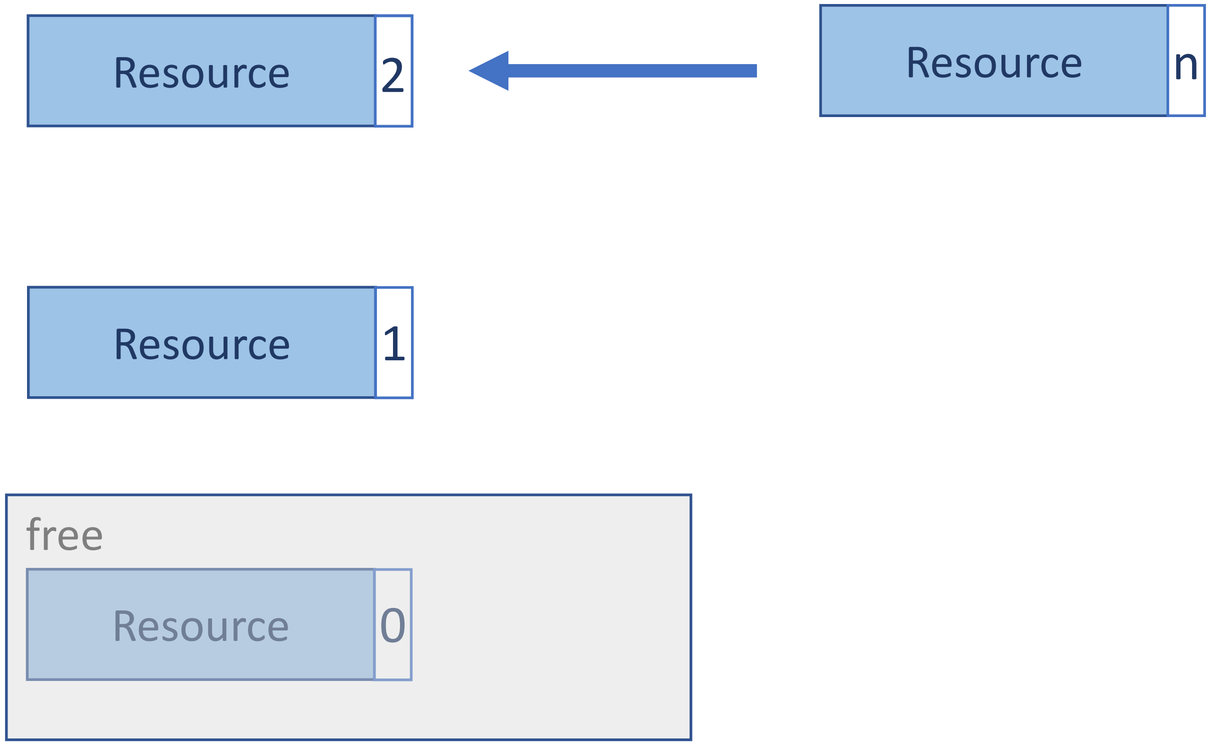
# Reference Counting 101



# Reference Counting 101

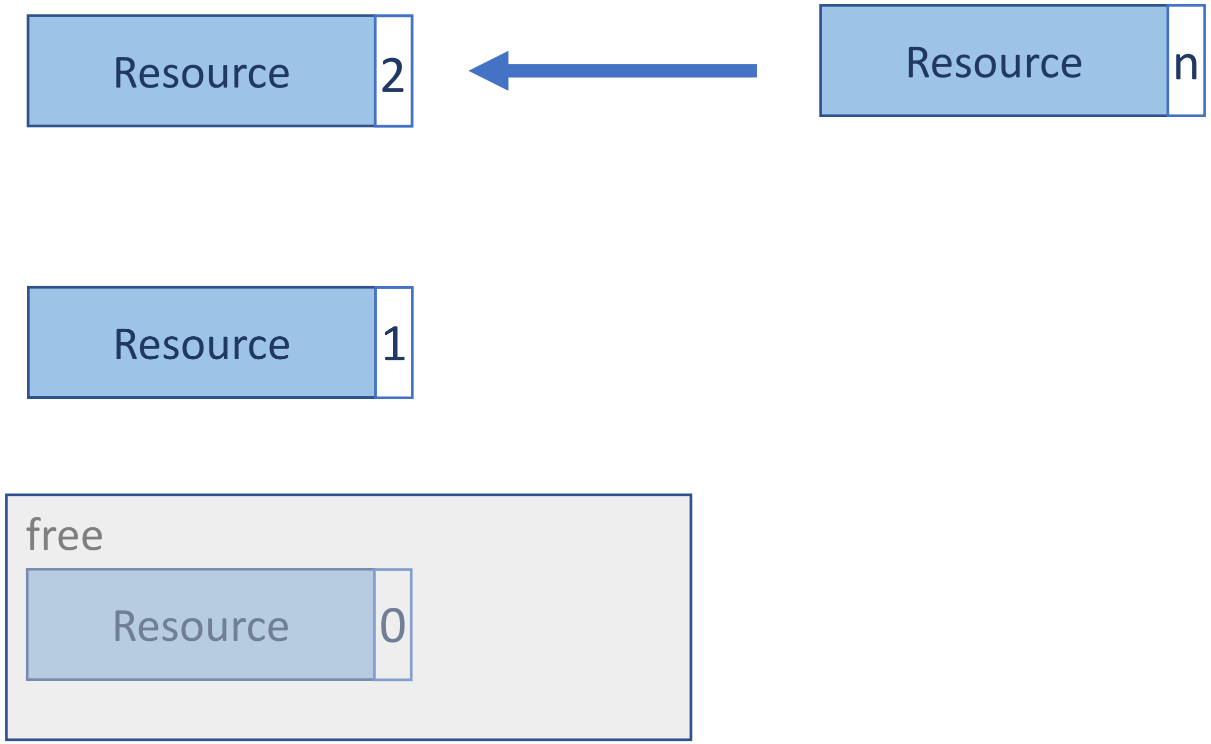


# Reference Counting 101



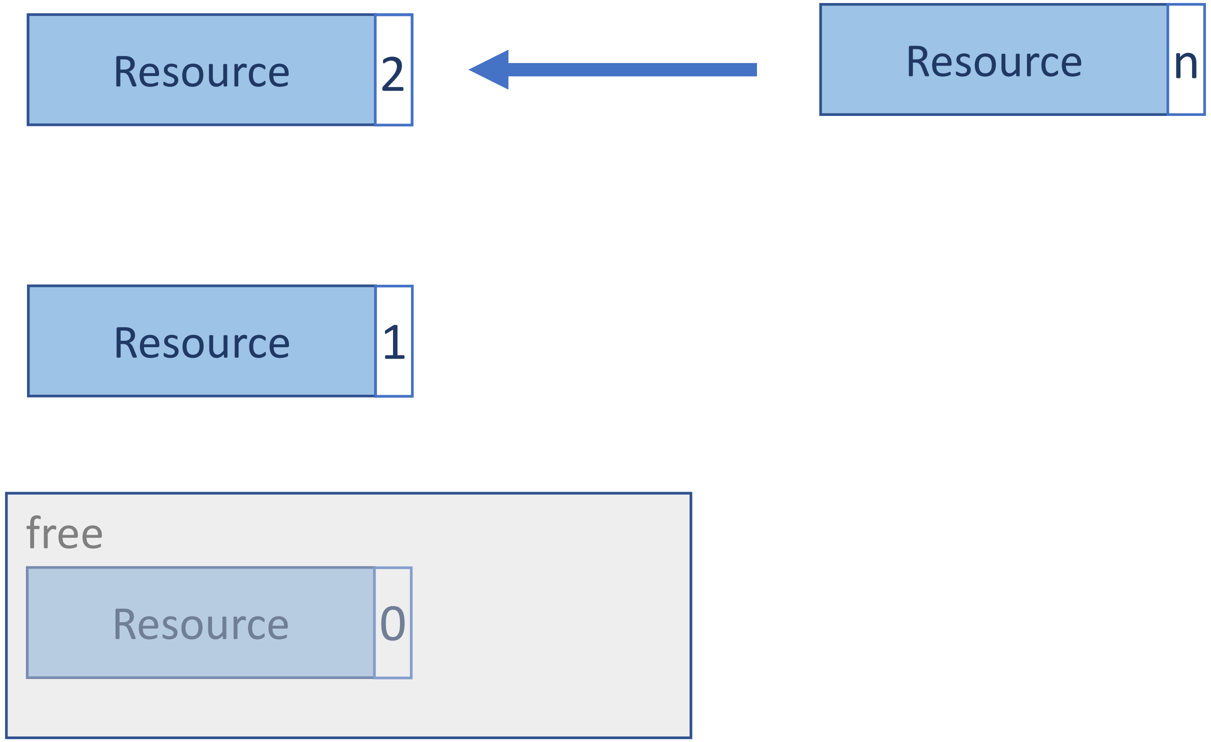

✓ Low memory overhead

# Reference Counting 101




- ✓ Low memory overhead
- ✓ Easy to implement

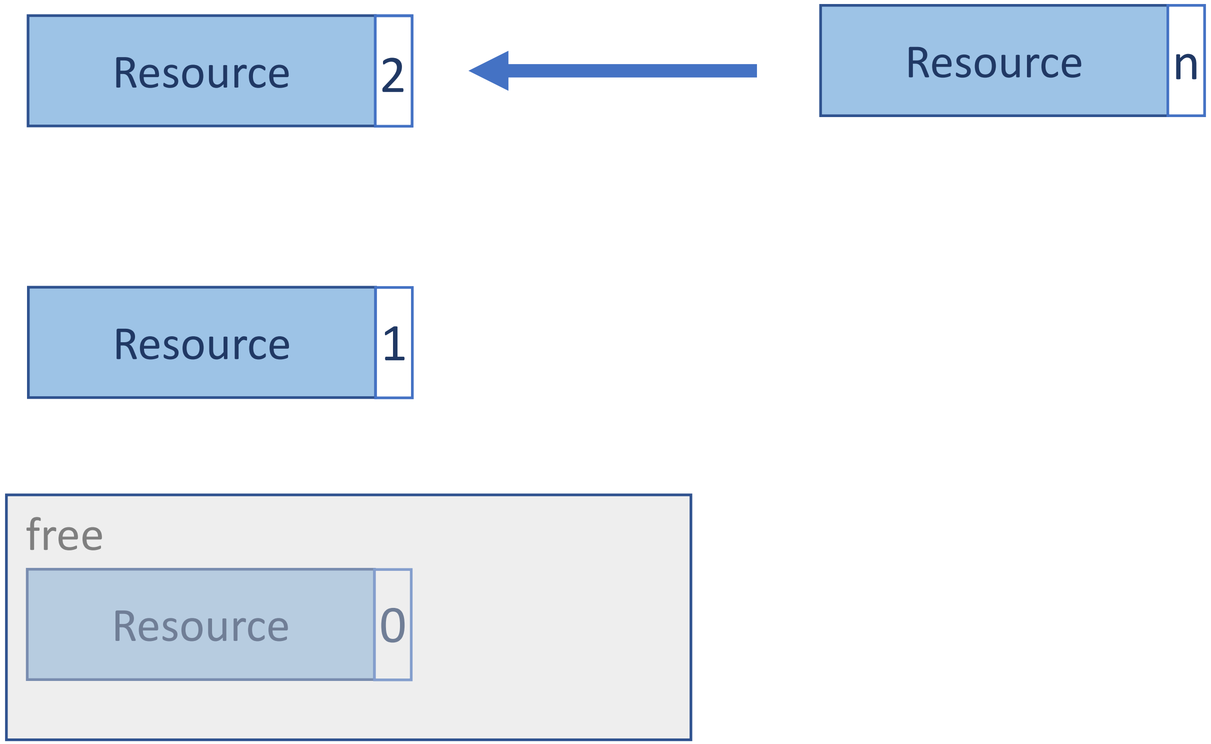
# Reference Counting 101




(shared_ptr⟨T⟩)

- ✓ Low memory overhead
- ✓ Easy to implement

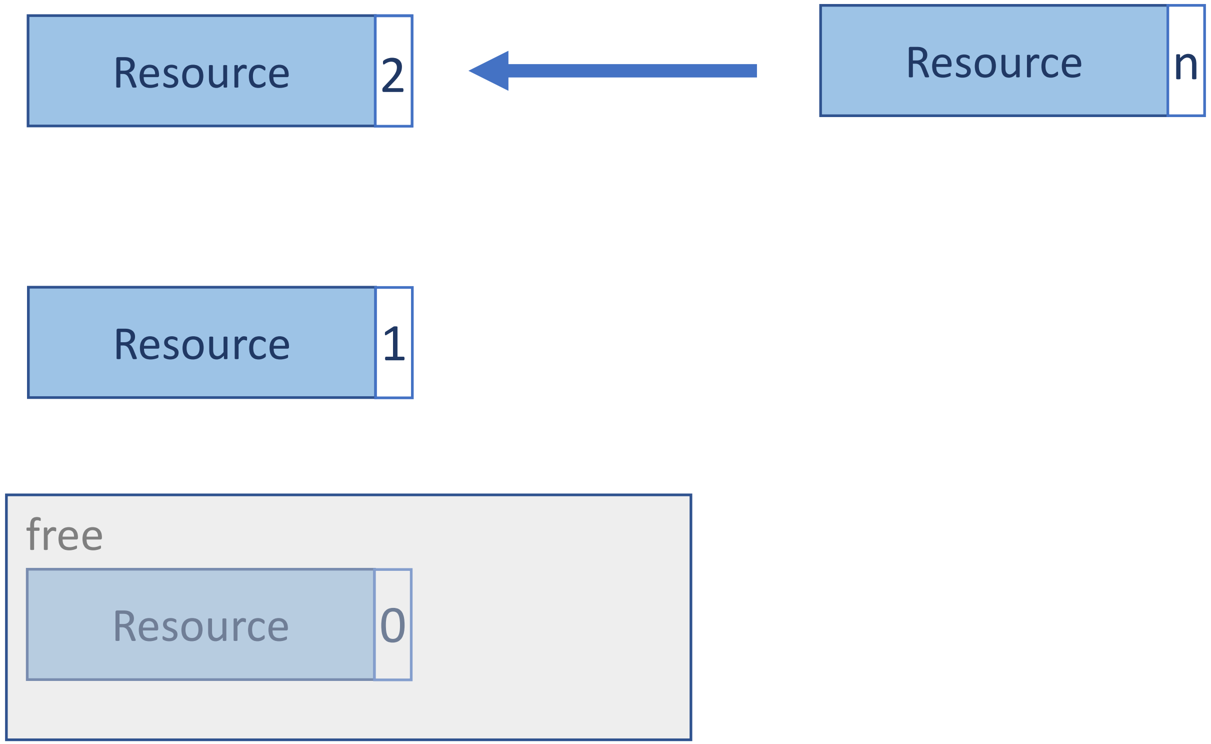
# Reference Counting 101




(shared_ptr⟨T⟩)

- ✓ Low memory overhead
- ✓ Easy to implement
- ▪ Precision

# Reference Counting 101

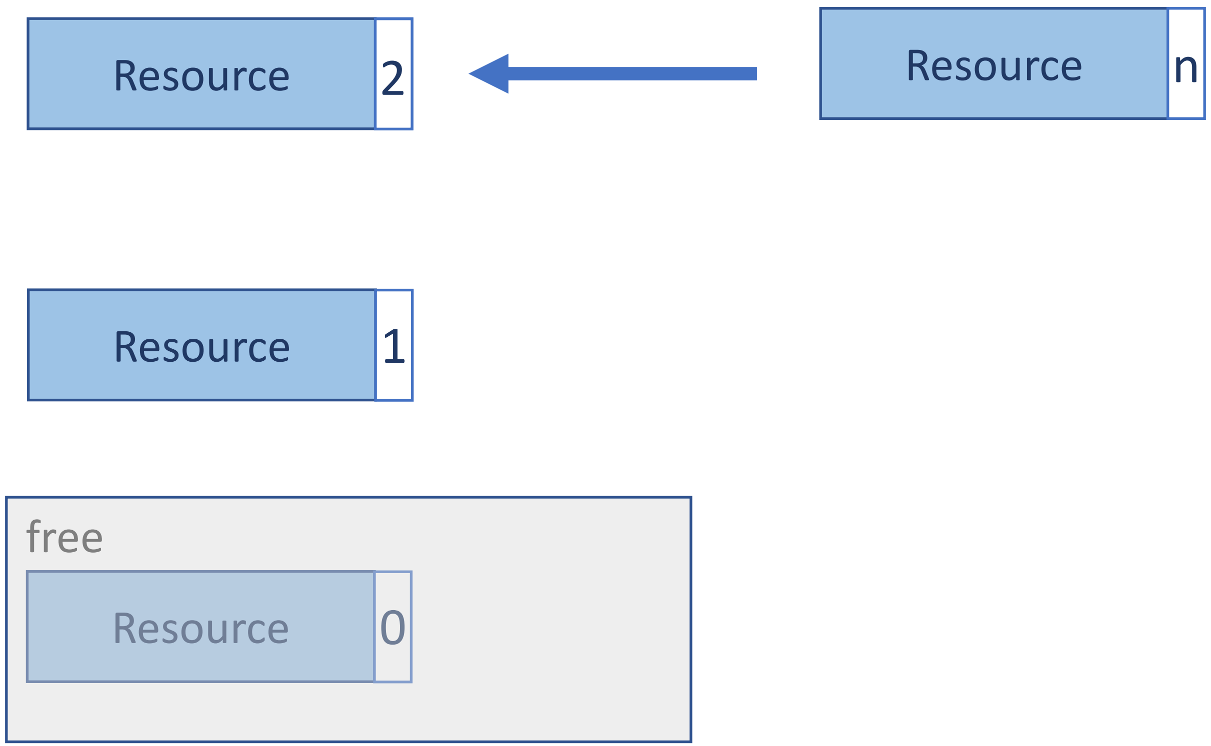



(shared_ptr⟨T⟩)

- ✓ Low memory overhead
- ✓ Easy to implement

- Precision
- Concurrency

# Reference Counting 101

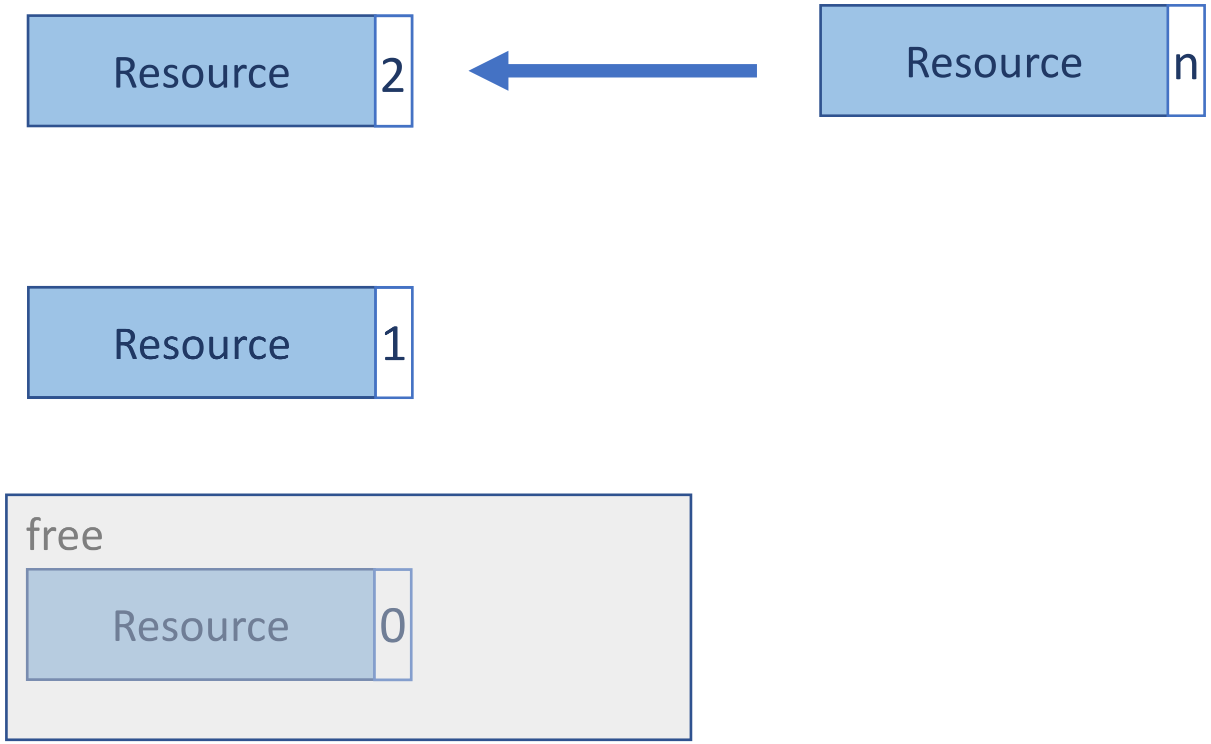



(shared_ptr⟨T⟩)

- ✓ Low memory overhead
- ✓ Easy to implement

- Precision
- Concurrency
- Cycles

# Reference Counting 101




(shared_ptr⟨T⟩)

- ✓ Low memory overhead
- ✓ Easy to implement

- Precision
- Concurrency
- Cycles

# Research Contributions

A programming language design that gives strong compile-time guarantees in order to enable efficient reference counting at run-time.

**Koka**

# Research Contributions

A programming language design that gives strong compile-time guarantees in order to enable efficient reference counting at run-time.

**Koka**

- Precision — this work
- Concurrency
- Cycles

# Agenda

① Perceus

② Koka 101

③ FBIP

Functional But In-Place

④ Linear Resource Calculus

$\lambda^1$

# Agenda

① Perceus

PrEcise Reference Counting with rEUse and Specialization

② Koka 101

③ FBIP

Functional But In-Place

④ Linear Resource Calculus

$\lambda^1$

# Agenda

① 

## Perceus

PrEcise Reference Counting with rEUse and Specialization

② 

## Koka 101

③ 

## FBIP

Functional But In-Place

④ 

## Linear Resource Calculus

$\lambda^1$

# Common reference counting implementations might retain memory longer than needed

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)
  val ys = map(xs, inc)
  print(ys)
}
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)
  print(ys)
}
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
}
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

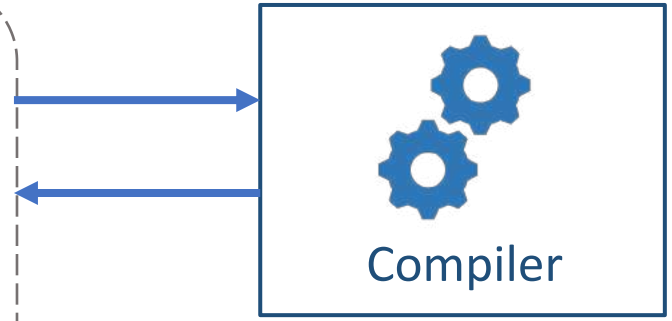
# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
}
```


```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```




```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000) // create large list
  val ys = map(xs, inc)     // increment elements
  print(ys)
  drop(xs)

  drop(ys)
}
```

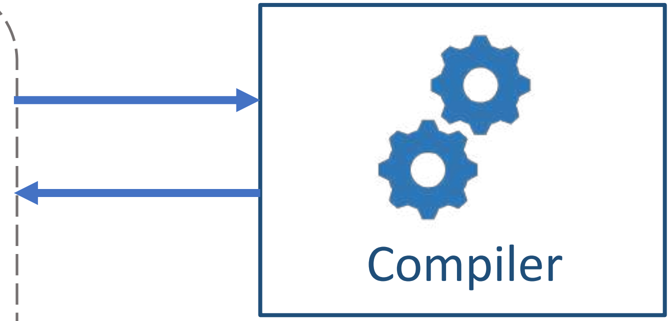
liveness of a reference is tied to its lexical scope


```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope




C++ (shared_ptr⟨T⟩)

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


C++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


C ++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)
Nim (finally)

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


C++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)
Nim (finally)
Swift

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

xs

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


C++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)
Nim (finally)
Swift

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

xs

xs ys

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


C ++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)
Nim (finally)
Swift

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

xs

xs ys

ys

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


C ++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)
Nim (finally)
Swift

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Common reference counting implementations might retain memory longer than needed

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

xs

xs ys

ys

liveness of a reference is tied to its lexical scope

**Can we do better?**

Compiler

C++ (shared_ptr⟨T⟩)
Rust (Rc⟨T⟩)
Nim (finally)
Swift

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Precise reference counting

```
fun foo() {
  val xs = list(1, 1000000) // create large list
  val ys = map(xs, inc)     // increment elements
  print(ys)
  drop(xs)
  drop(ys)
}
```

liveness of a reference is tied to its lexical scope


```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Precise reference counting

```
fun foo() {
  val xs = list(1, 1000000)  // create large list
  val ys = map(xs, inc)      // increment elements
  drop(xs)
  print(ys)
  drop(ys)
}
```

drop resources as soon as possible


```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Perceus passes ownership of references

```
fun foo() {
  val xs = list(1, 1000000) // create large list
  val ys = map(xs, inc)     // increment elements
  drop(xs)
  print(ys)
  drop(ys)
}
```

ownership


```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

# Perceus passes ownership of references

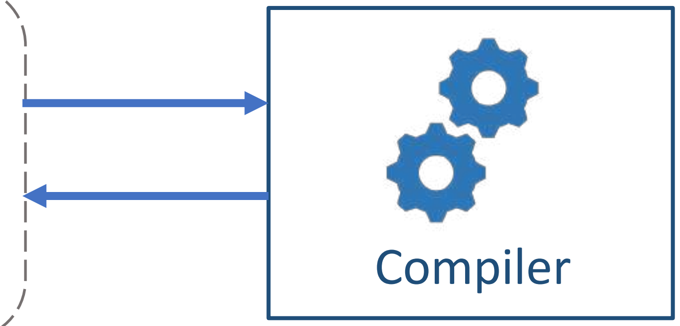
```
fun foo() {
  val xs = list(1, 1000000) // create large list
  val ys = map(xs, inc)     // increment elements
  print(ys)
}
```

ownership


```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```

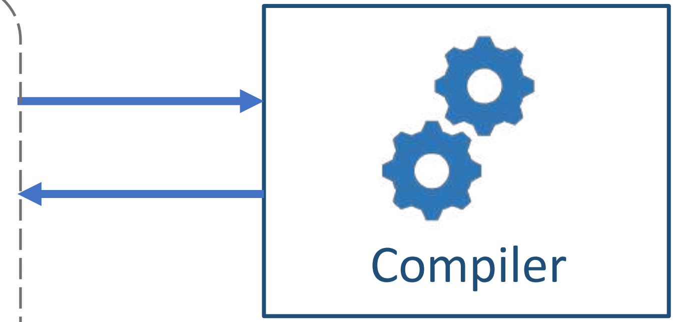
# Perceus passes ownership of references

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) -> Cons(f(x), map(xx, f))
    Nil         -> Nil
  }
}
```



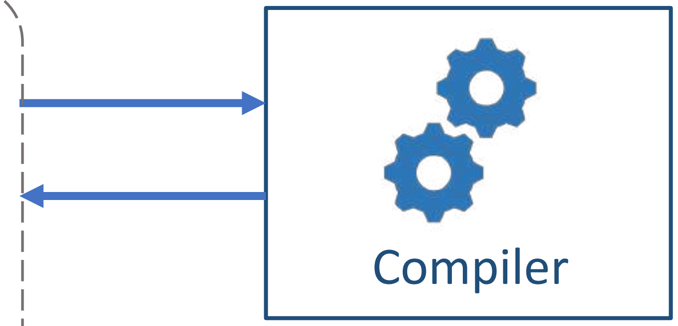
# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      Cons(f(x), map(xx, f))
    }
    Nil {
      Nil
    }
  }
}
```



# Precise reference counting

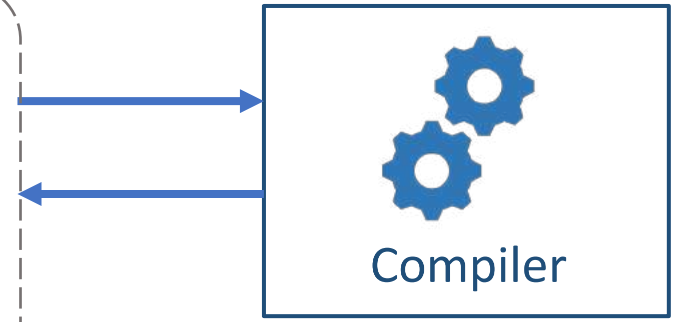
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      Cons(f(x), map(xx, f))
    }
    Nil {
      Nil
    }
  }
}
```




1. dup/drop insertion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion

# Precise reference counting

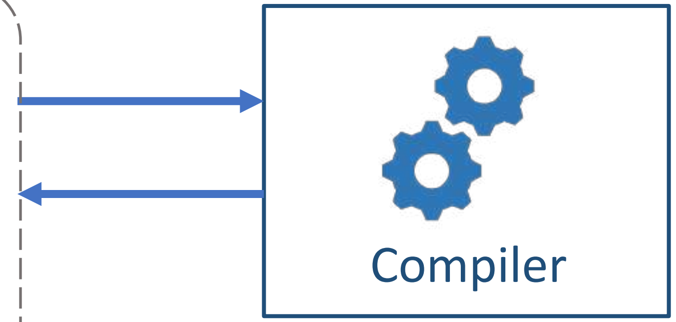
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

returns itself


1. dup/drop insertion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

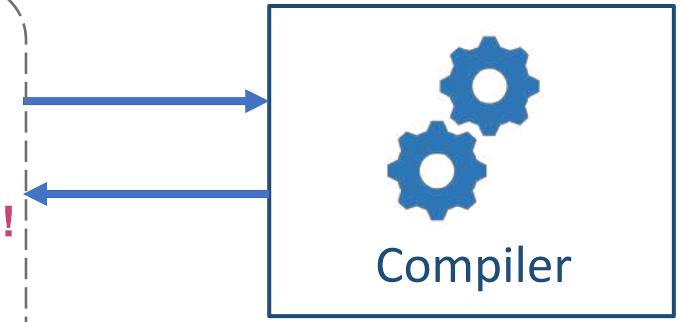



1. dup/drop insertion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

the memory usage is halved!



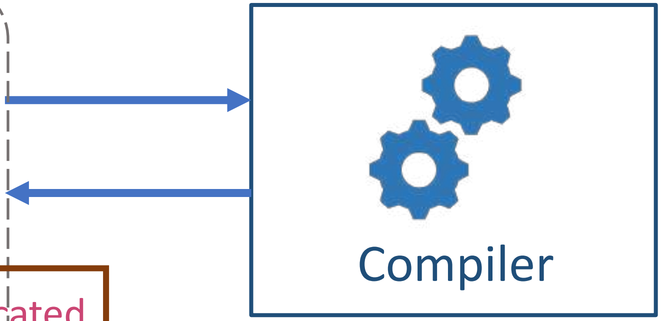

1. dup/drop insertion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

**the memory usage is halved!**

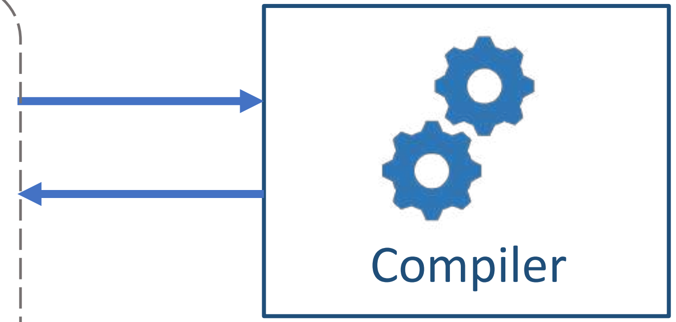
the list xs is deallocated while the new list is being allocated.




1. dup/drop insertion

# Precise reference counting

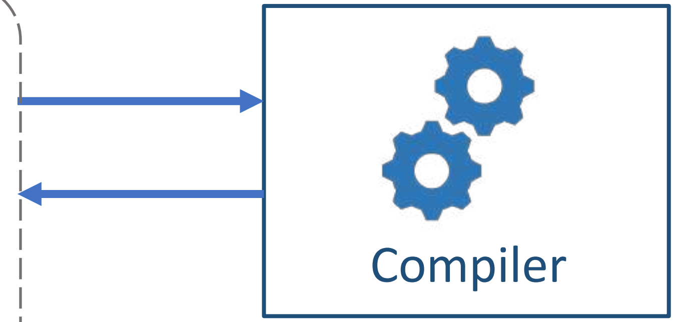
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

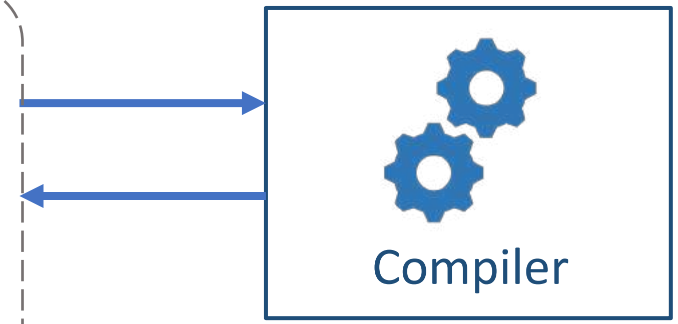



1. dup/drop insertion

2. drop specialization

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion
2. drop specialization

```
fun drop( x ) {
  if (is-unique(x))
  then drop children of x;
       free(x)
  else decref(x)
}
```

# Precise reference counting

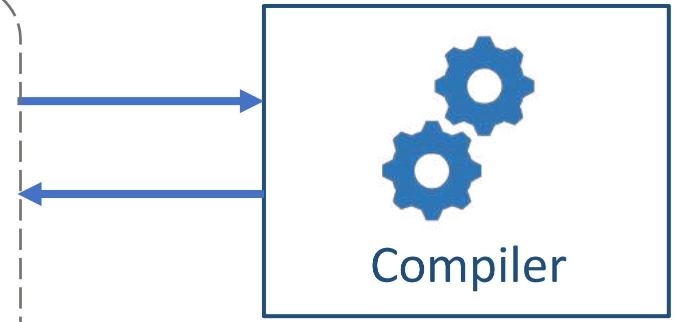
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```


1. dup/drop insertion
2. drop specialization

```
fun drop( x ) {
  if (is-unique(x))
  then drop children of x;
       free(x)
  else decref(x)
}
```

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then drop(x); drop(xx); free(xs);
      else decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```



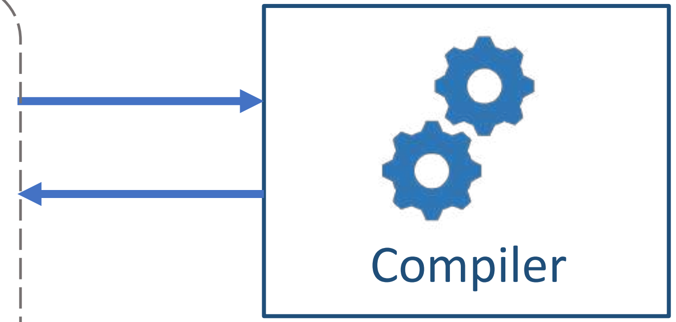

1. dup/drop insertion

2. drop specialization

```
fun drop( x ) {
  if (is-unique(x))
  then drop children of x;
       free(x)
  else decref(x)
}
```

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then drop(x); drop(xx); free(xs);
      else decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

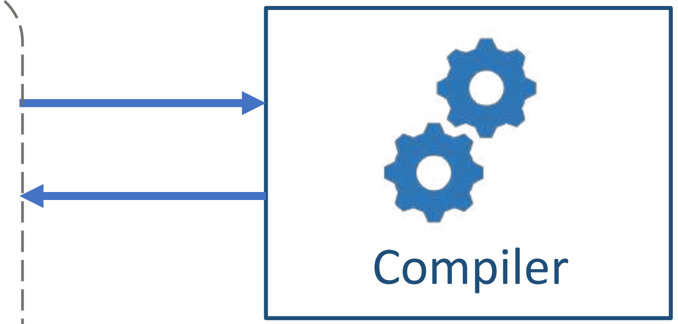
# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then drop(x); drop(xx); free(xs);
      else decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```


1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

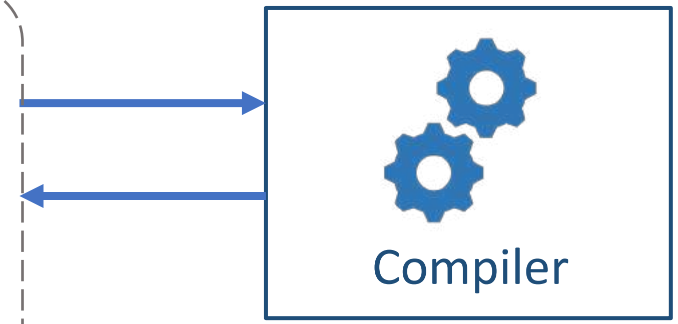
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then drop(x); drop(xx); free(xs);
      else decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then drop(x); drop(xx); free(xs);
      else decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then free(xs);
      else decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```


1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

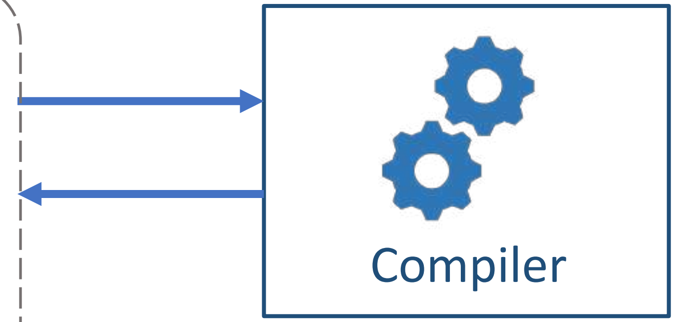
# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      if (is-unique(xs))
      then free(xs);
      else dup(x); dup(xx); decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```


1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

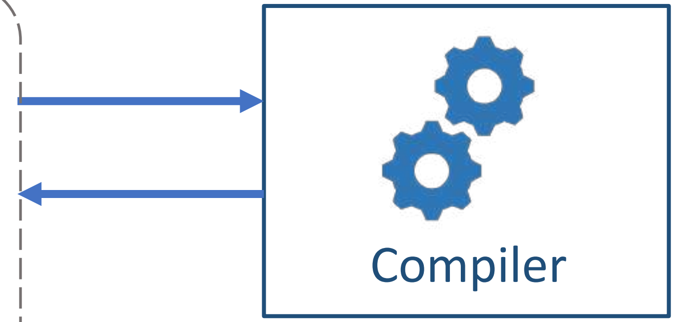
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {

      if (is-unique(xs))
      then free(xs);
      else dup(x); dup(xx); decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      if (is-unique(xs))
      then free(xs);
      else dup(x); dup(xx); decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

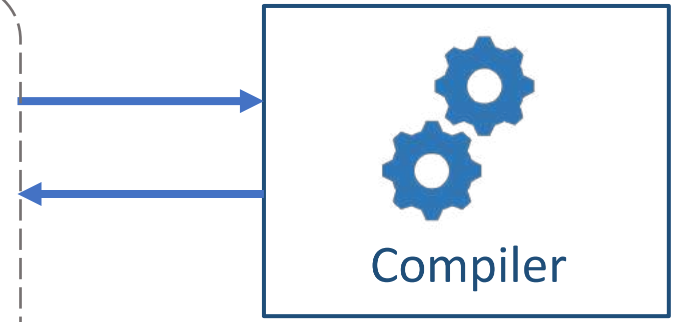



1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      if (is-unique(xs))
      then free(xs);
      else dup(x); dup(xx); decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

fast path



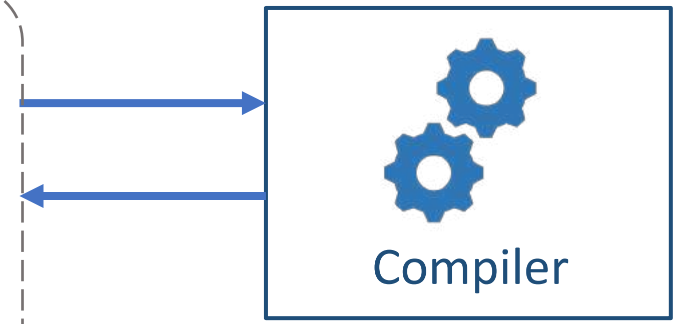

1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      if (is-unique(xs))
      then free(xs);
      else dup(x); dup(xx); decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

fast path

free xs and immediately allocate a fresh Cons node




1. dup/drop insertion
2. drop specialization
3. push down dup and fusion

# Precise reference counting

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      if (is-unique(xs))
      then free(xs);
      else dup(x); dup(xx); decref(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

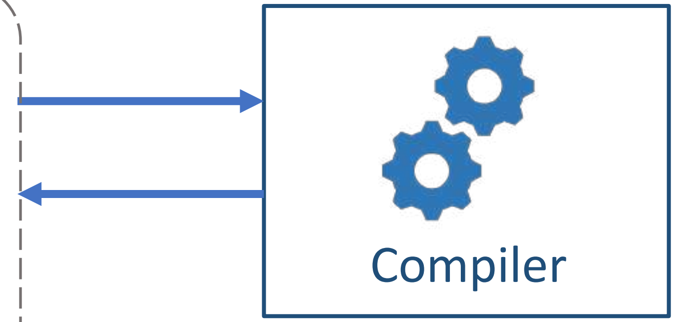
fast path

free xs and immediately allocate a fresh Cons node


1. dup/drop insertion

# Precise reference counting

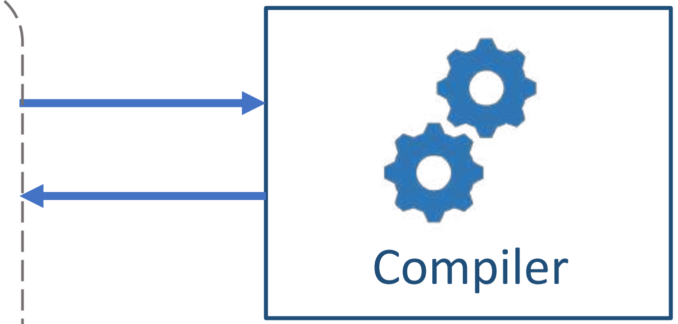
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion

# Precise reference counting with reuse

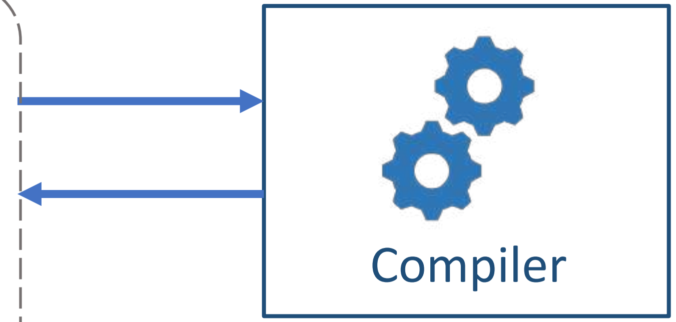
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion

# Precise reference counting with reuse

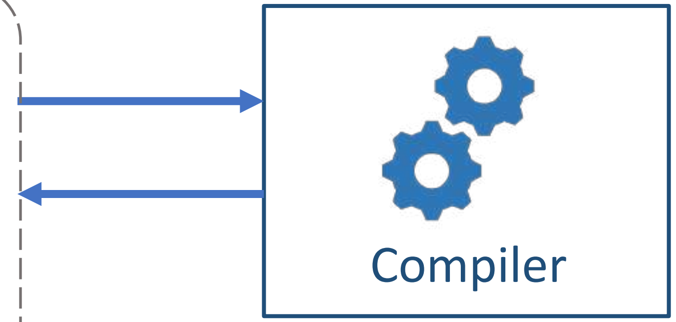
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion /reuse analysis

# Precise reference counting with reuse

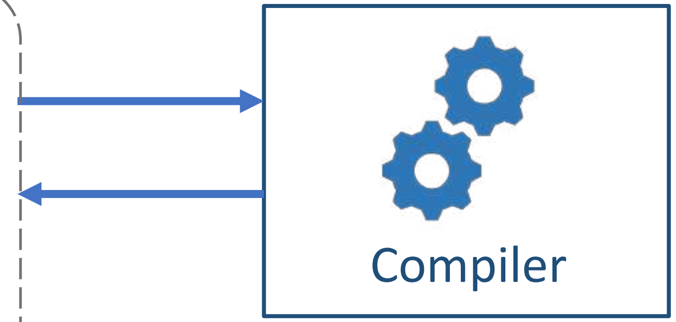
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion /reuse analysis

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion /reuse analysis

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

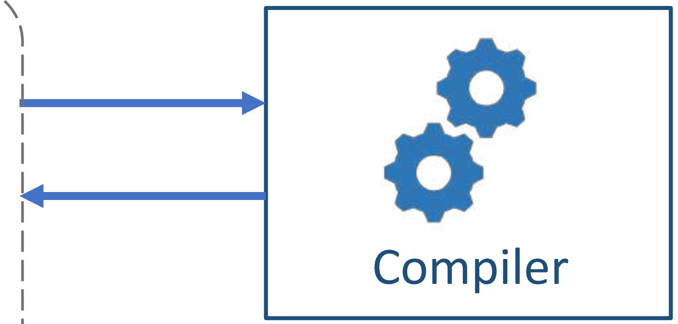
try to reuse
xs directly


1. dup/drop insertion /reuse analysis

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

try to reuse
xs directly

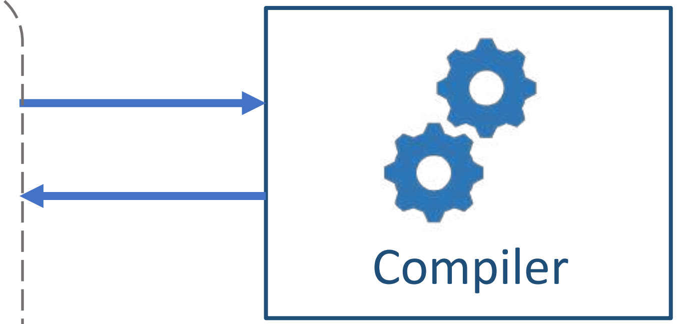



1. dup/drop insertion /reuse analysis
2. drop-reuse specialization

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

try to reuse xs directly




1. dup/drop insertion /reuse analysis
2. drop-reuse specialization

```
fun drop( x ) {
  if (is-unique(x))
  then drop children of x;
       free(x)
  else decref(x)
}
```

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

try to reuse
xs directly


1. dup/drop insertion /reuse analysis

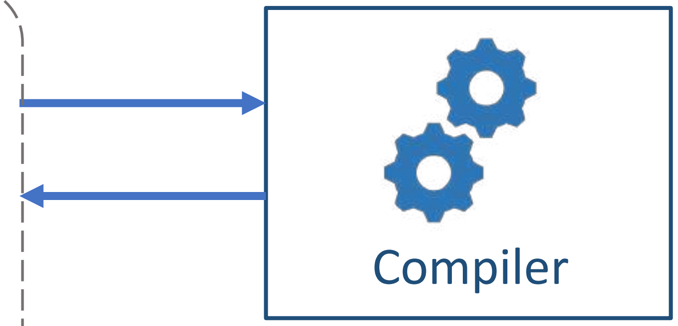
2. drop-reuse specialization

```
fun drop-reuse( x ) {
  if (is-unique(x))
  then drop children of x;
       & x
  else decref(x) ; Null
}
```

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs);
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

try to reuse
xs directly



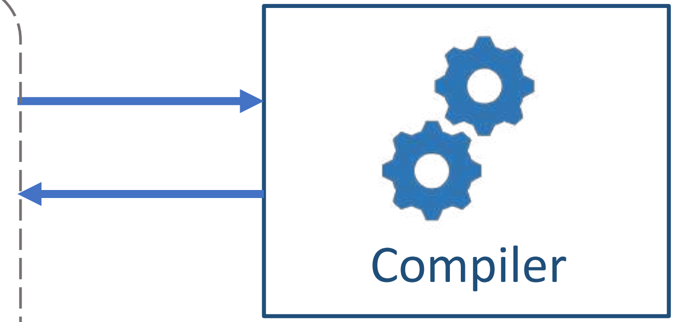

1. dup/drop insertion /reuse analysis

2. drop-reuse specialization

```
fun drop-reuse( x ) {
  if (is-unique(x))
  then drop children of x;
       & x // returns the address of x
  else decref(x) ; Null
}
```

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = if (is-unique(xs))
                 then drop(x); drop(xx); &xs
                 else decref(xs); Null
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion /reuse analysis

2. drop-reuse specialization

```
fun drop-reuse( x ) {
  if (is-unique(x))
  then drop children of x;
       & x // returns the address of x
  else decref(x) ; Null
}
```

# Precise reference counting with reuse

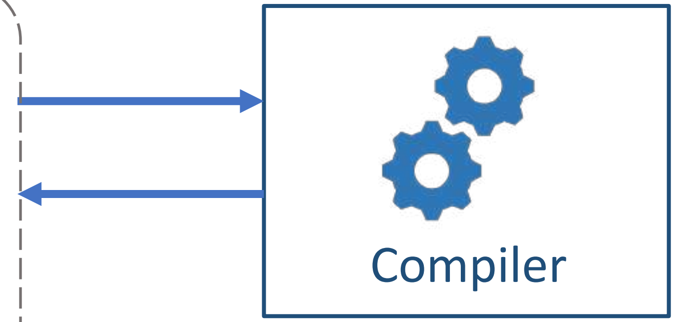
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = if (is-unique(xs))
                 then drop(x); drop(xx); &xs
                 else decref(xs); Null
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```


1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion

```
fun drop-reuse( x ) {
  if (is-unique(x))
  then drop children of x;
       & x // returns the address of x
  else decref(x) ; Null
}
```

# Precise reference counting with reuse

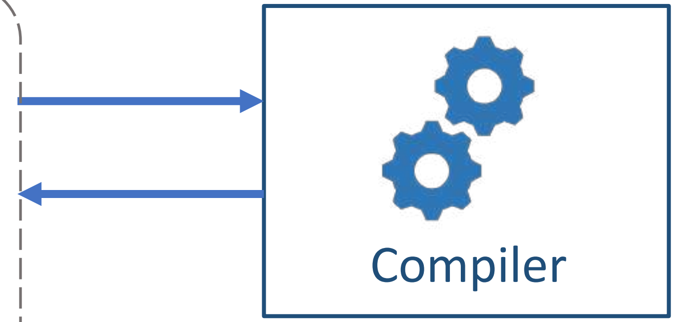
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                 then &xs;
                 else dup(x); dup(xx);
                      decref(xs); Null
      Cons@ru(dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```




1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                 then &xs;                fast path
                 else dup(x); dup(xx);
                      decref(xs); Null
      Cons@ru(dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```



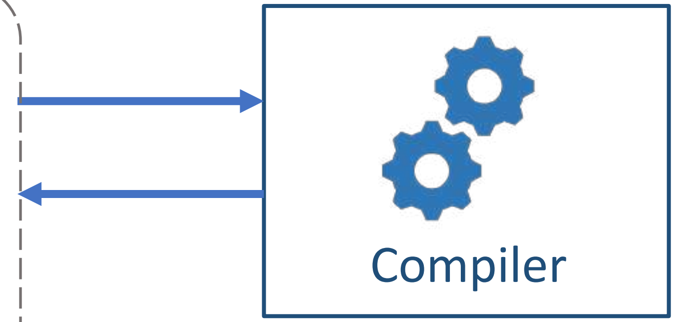

1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru(dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

fast path

reuse




1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion

# Precise reference counting with reuse

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx); drop(xs);



      Cons( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
               then &xs;
               else dup(x); dup(xx);
                    decref(xs); Null
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

fast path

reuse

# Agenda

① Perceus

② Koka 101

③ Functional But In-Place (FBIP)

④ Linear Resource Calculus

$\lambda^1$

# Agenda

① Perceus

② Koka 101

③ Functional But In-Place (FBIP)

④ Linear Resource Calculus

$\lambda^1$

# Koka tracks all (side) effects using algebraic effects

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
    m / n
}
```

div by zero!

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

effect

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

effect

operation

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

operation

effect

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

operation

effect

track effects in types

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

effect type system

operation

effect

track effects in types

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

operation

effect

**effect type system**

track effects in types

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

```
fun div1(m, n) {
  with handler {
    fail() { Nothing }
  }
  Just(div(m, n))
}
```

```
fun div2(m, n){
  with handler {
    fail() { resume(0) }
  }
  div(m, n)
}
```

```
fun div3(m, n){
  with handler {
    fail() { resume (0) + (resume (0) }
  }
  div(m, n)
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

operation

effect

**effect type system**

track effects in types

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

effect handler

```
fun div1(m, n) {
  with handler {
    fail() { Nothing }
  }
  Just(div(m, n))
}
```

```
fun div2(m, n){
  with handler {
    fail() { resume(0) }
  }
  div(m, n)
}
```

```
fun div3(m, n){
  with handler {
    fail() { resume (0) + (resume (0) }
  }
  div(m, n)
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

operation

effect

effect type system

track effects in types

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

effect handler

```
fun div1(m, n) {
  with handler {
    fail() { Nothing }
  }
  Just(div(m, n))
}
```

resume with default value

```
fun div2(m, n){
  with handler {
    fail() { resume(0) }
  }
  div(m, n)
}
```

```
fun div3(m, n){
  with handler {
    fail() { resume (0) + (resume (0) }
  }
  div(m, n)
}
```

# Koka tracks all (side) effects using algebraic effects

```
div(m : int, n : int ) : exn int {
   m / n
}
```

operation

effect

**effect type system**

track effects in types

```
effect exn {
  fail() : int
}

div(m : Int, n : int) : exn int {
  if (n == 0) then fail ()
  else m / n
}
```

perform an operation

effect handler

```
fun div1(m, n) {
  with handler {
    fail() { Nothing }
  }
  Just(div(m, n))
}
```

resume with default value

```
fun div2(m, n){
  with handler {
    fail() { resume(0) }
  }
  div(m, n)
}
```

resume multiple times

```
fun div3(m, n){
  with handler {
    fail() { resume (0) + (resume (0) }
  }
  div(m, n)
}
```

# Reference counting with strong static guarantees

**With such a strong effect type system …**

1. Non-linear control flow
2. Concurrency
3. Mutation / cycles

# Reference counting with strong static guarantees

**With such a strong effect type system …**

1. Non-linear control flow
2. Concurrency
3. Mutation / cycles

**Goal**: mitigate the impact of concurrency and cycles.

# Reference counting with strong static guarantees

**With such a strong effect type system …**

1. Non-linear control flow
2. Concurrency
3. Mutation / cycles

**Goal**: mitigate the impact of concurrency and cycles.

**Non-goal:** a general solution to all problems with reference counting.

# Non-linear control flow

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru(dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

# Non-linear control flow

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru (dup(f) (x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

f raises an exception!

# Non-linear control flow

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

f raises an exception!

xx and f would leak and never be dropped

1

# Non-linear control flow

```
f : a -> exn b
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                 then &xs;
                 else dup(x); dup(xx);
                      decref(xs); Null
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

f raises an exception!

xx and f would leak and never be dropped

# Non-linear control flow

```
f : a -> exn b
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                 then &xs;
                 else dup(x); dup(xx);
                      decref(xs); Null
      match(dup(f)(x)) {
        Error(err) { drop(xx); drop(f); Error(err); }
        Ok(y) { match(map(xx, f)) {
           Error(err) -> drop(y); Error(err);
           Ok(ys)     -> Cons(y, ys);
          }
        }
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

1

# Non-linear control flow

```
f : a -> exn b
```

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                    then &xs;
                    else dup(x); dup(xx);
                         decref(xs); Null
      match(dup(f) (x)) {
        Error(err) { drop(xx); drop(f); Error(err); }
        Ok(y) { match(map(xx, f)) {
           Error(err) -> drop(y); Error(err);
           Ok(ys)     -> Cons(y, ys);
          }
        }
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

all control-flow is compiled to explicit control-flow

1

# Non-linear control flow

```
f : a -> exn b
```

effects can also be polymorphic

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                 then &xs;
                 else dup(x); dup(xx);
                      decref(xs);  Null
      match( dup(f) (x)) {
        Error(err) { drop(xx);  drop(f);  Error(err); }
        Ok(y) { match(map(xx, f)) {
           Error(err) -> drop(y);  Error(err);
           Ok(ys)     -> Cons(y, ys);
          }
        }
    }
    Nil {
      drop(xs);  drop(f);
      Nil
    }
  }
}
```

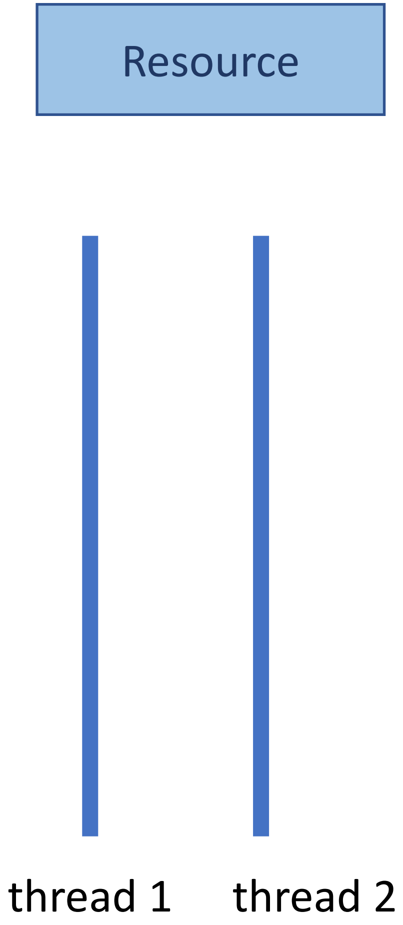
all control-flow is compiled to explicit control-flow

2

# Concurrency

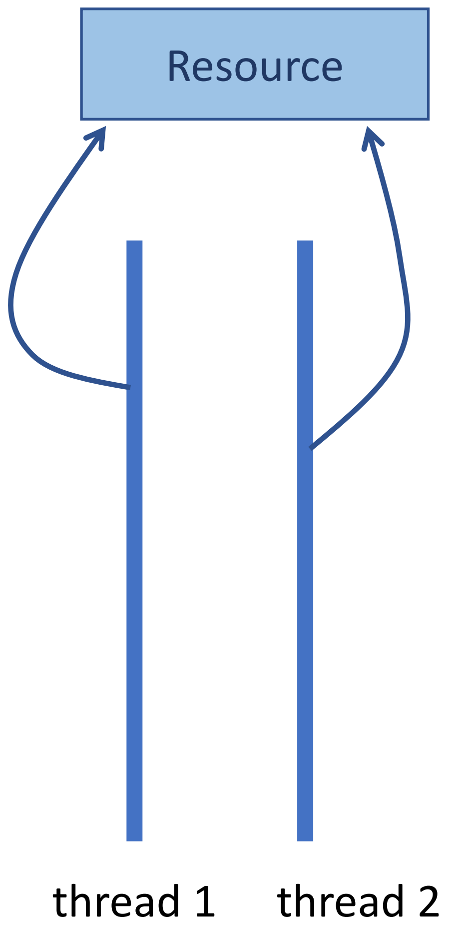
Resource

# Concurrency

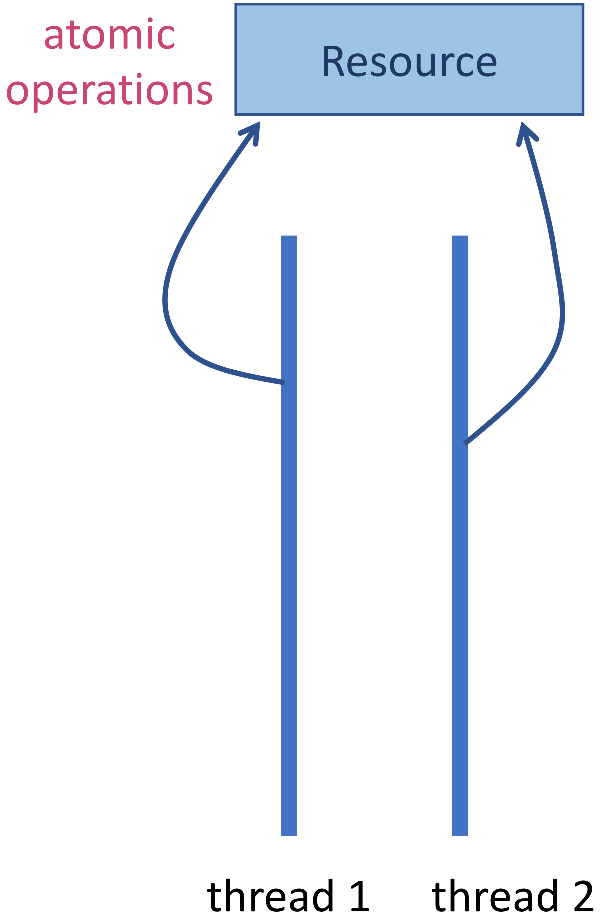


2

# Concurrency

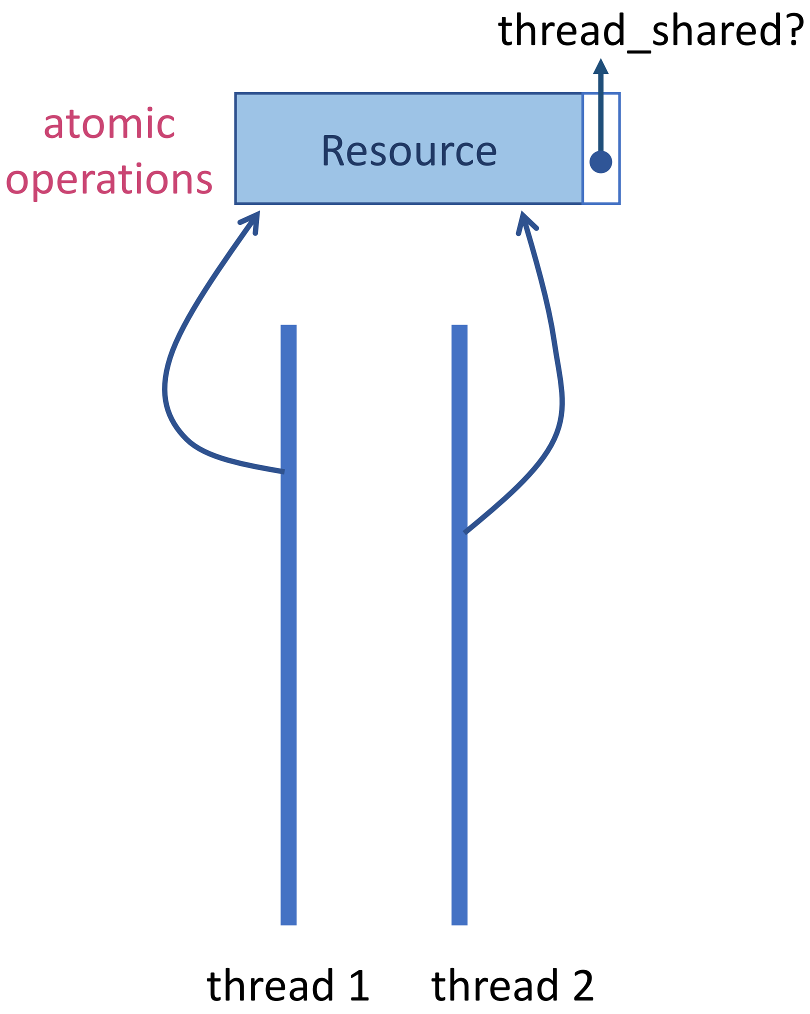


# Concurrency

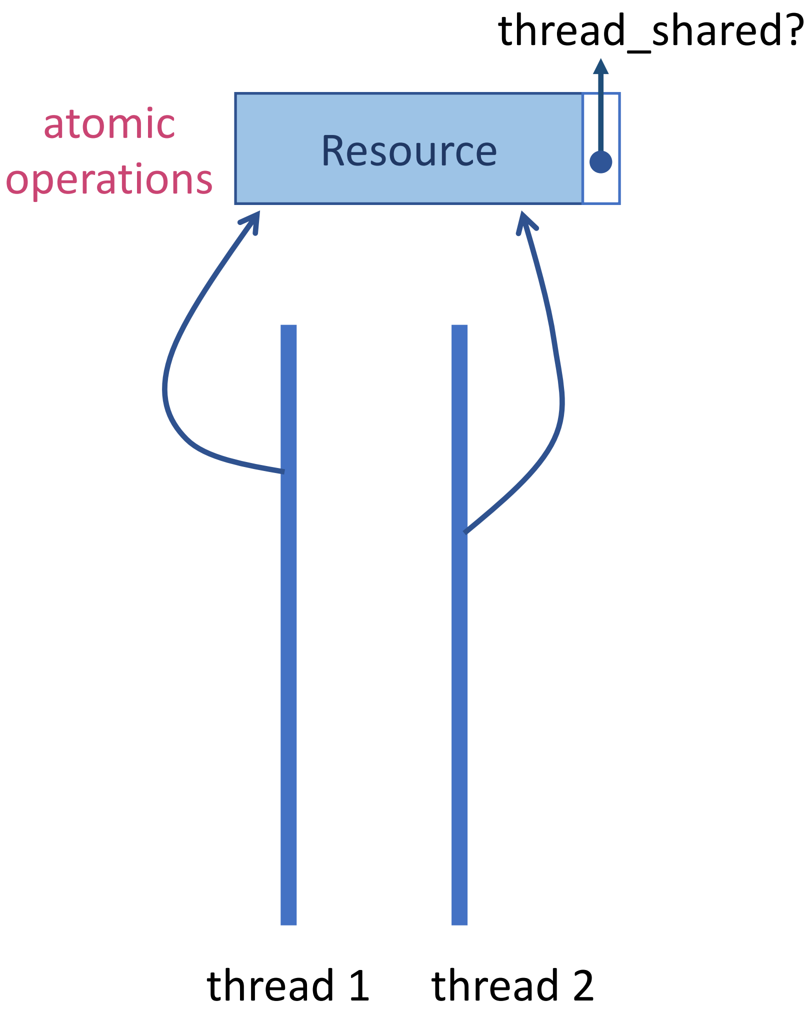


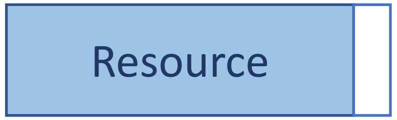
2

# Concurrency



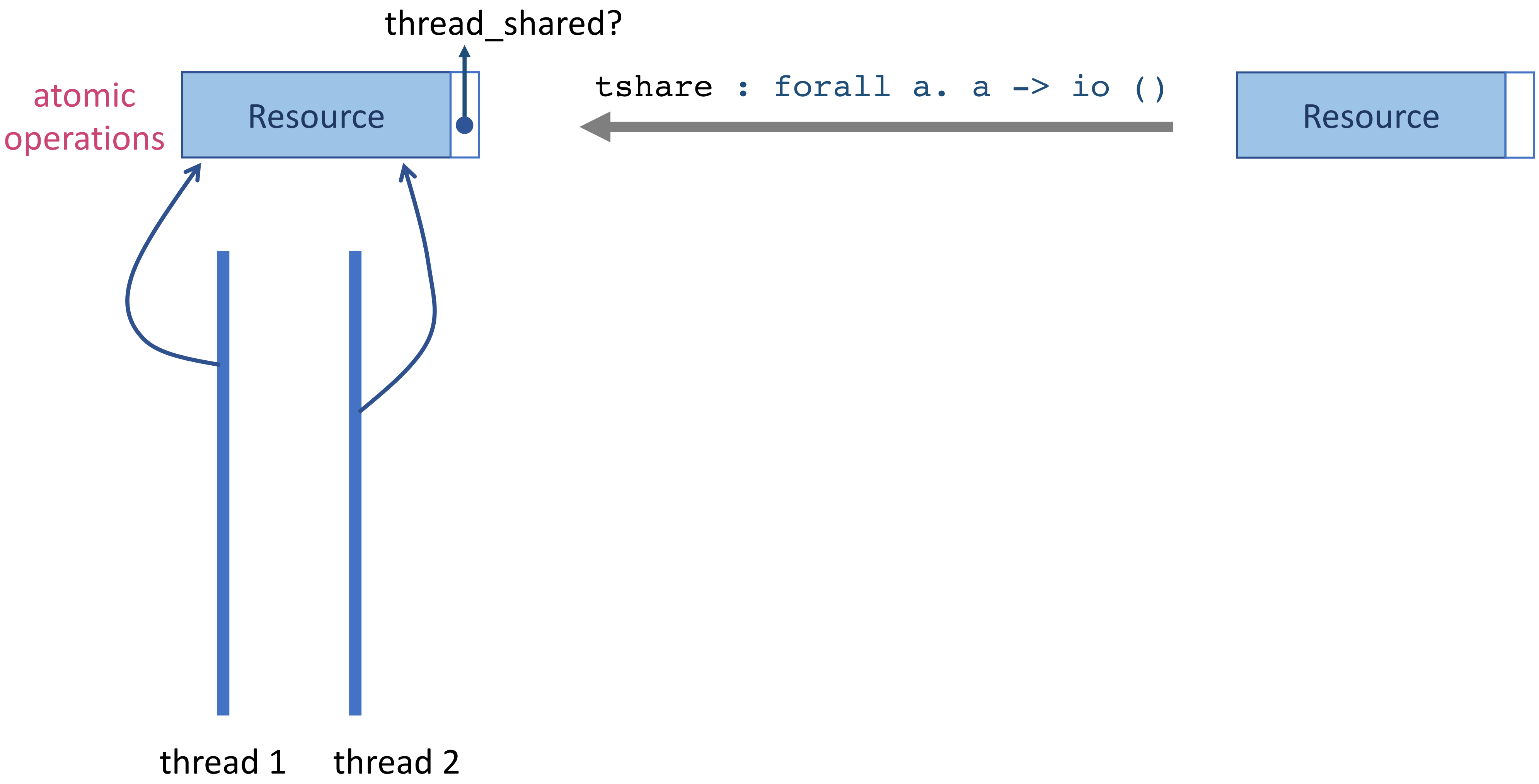
# Concurrency



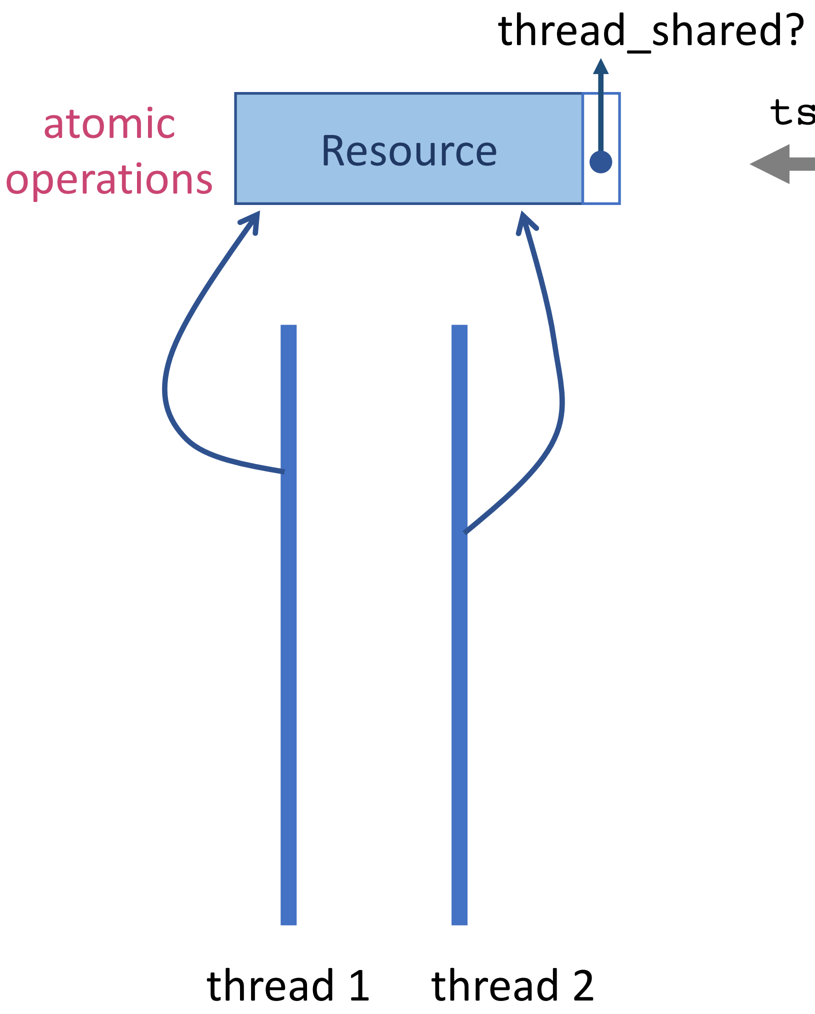



# Concurrency

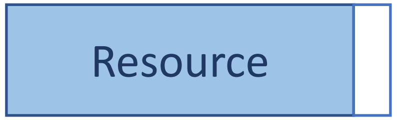
2



# Concurrency

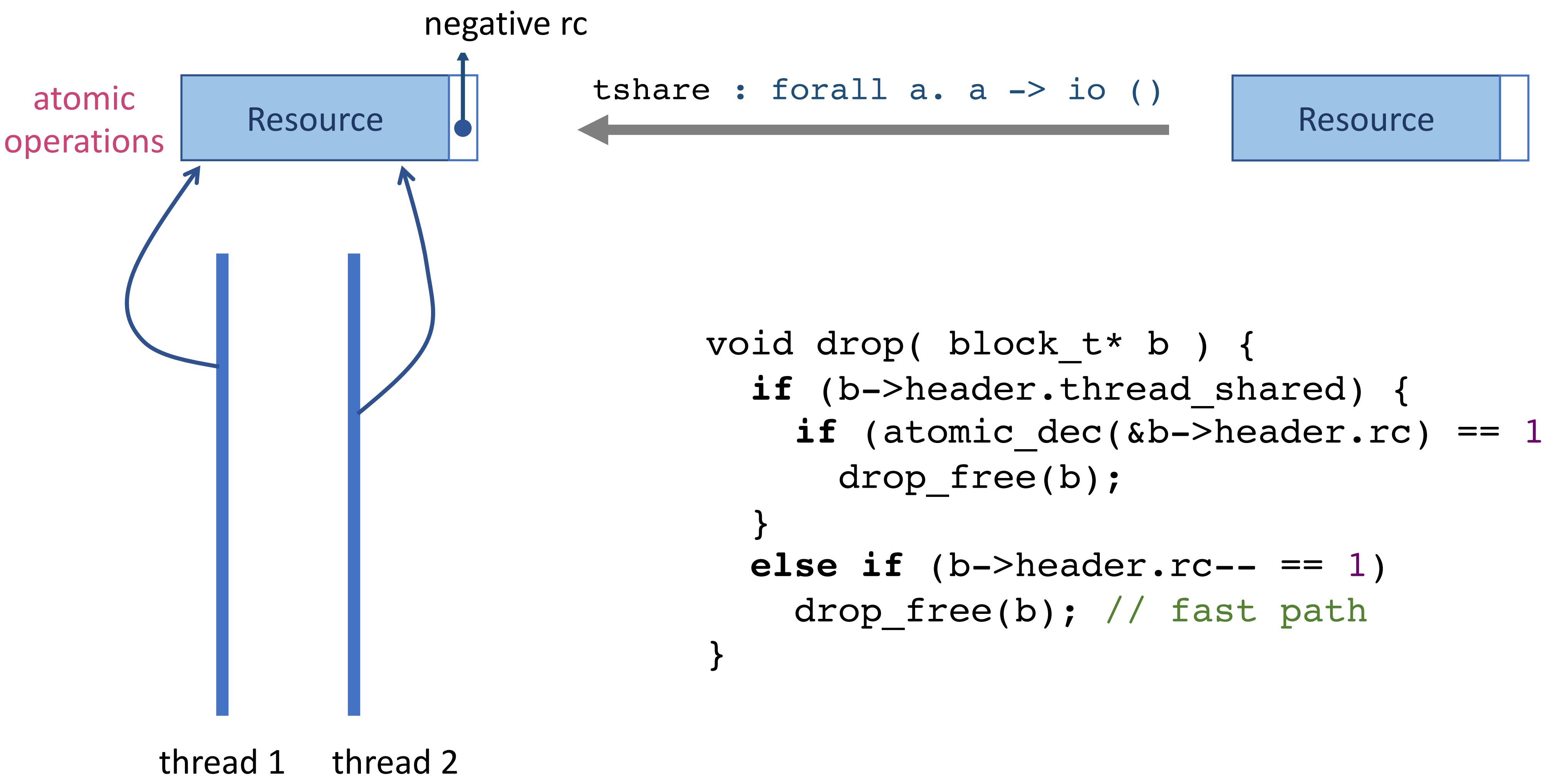



```
tshare : forall a. a -> io ()
```




```
void drop( block_t* b ) {
  if (b->header.thread_shared) {
    if (atomic_dec(&b->header.rc) == 1)
      drop_free(b);
  }
  else if (b->header.rc-- == 1)
    drop_free(b); // fast path
}
```

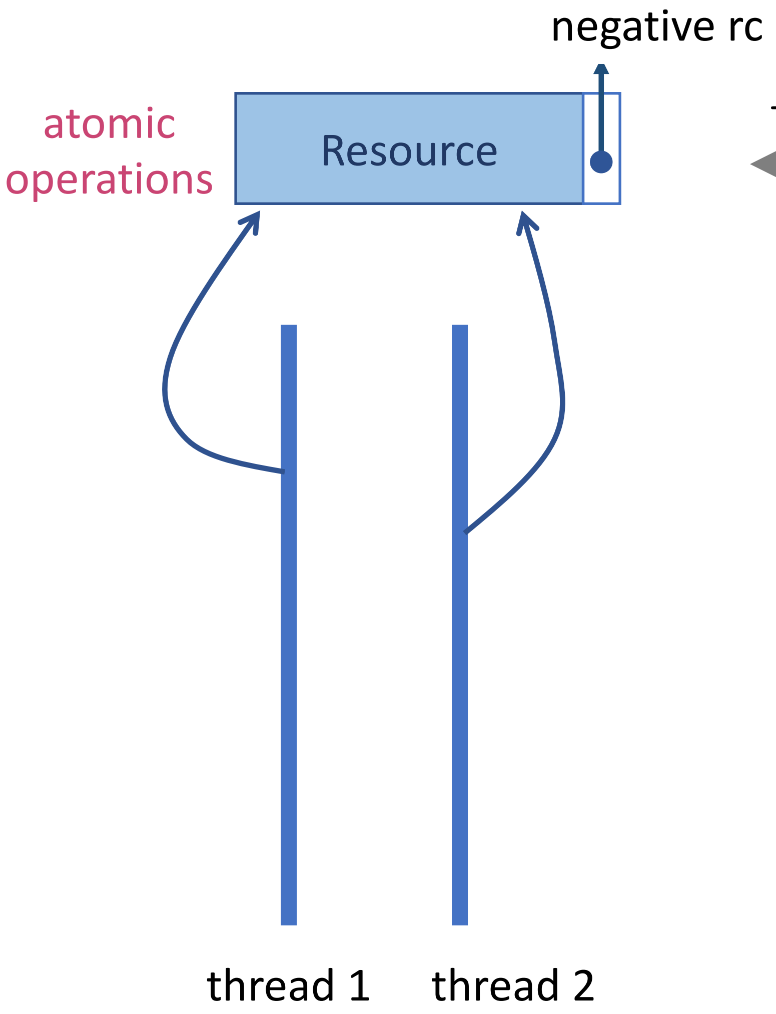
# Concurrency



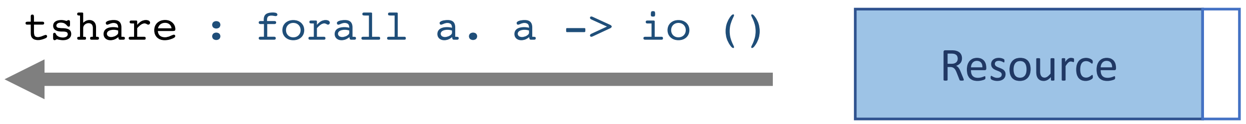

```
void drop( block_t* b ) {
  if (b->header.thread_shared) {
    if (atomic_dec(&b->header.rc) == 1)
      drop_free(b);
  }
  else if (b->header.rc-- == 1)
    drop_free(b); // fast path
}
```

2

# Concurrency




```
tshare : forall a. a -> io ()
```




```
void drop( block_t* b ) {
  if (b->header.rc <= 1)
    drop_check(b);
  else
    b->header.rc--;
}
```

3

# Mutation and cycles

# Mutation and cycles

create a mutable reference cell

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

3

# Mutation and cycles

create a mutable reference cell

stateful effect

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

3

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

# Mutation and cycles



create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

3

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

```
fun (:=)( r : ref⟨h,a⟩, x : a ) : st⟨h⟩ ()
{
  val y = r->value
  r->value := x
  drop(y)
}
```

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

```
fun (:=)( r : ref⟨h,a⟩, x : a ) : st⟨h⟩ ()
{
  val y = r->value
  r->value := x
  drop(y)
}
```

3

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

```
fun (:=)( r : ref⟨h,a⟩, x : a ) : st⟨h⟩ ()
{
  val y = r->value
  r->value := x
  drop(y)
}
```

3

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

dup a freed object!

```
fun (:=)( r : ref⟨h,a⟩, x : a ) : st⟨h⟩ ()
{
  val y = r->value
  r->value := x
  drop(y)
}
```

3

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

dup a freed object!

```
fun (:=)( r : ref⟨h,a⟩, x : a ) : st⟨h⟩ ()
{
  val y = r->value
  r->value := x
  drop(y)
}
```

- **FBIP**: Functional but in-place

3

# Mutation and cycles

create a mutable reference cell

stateful effect

first-class value

```
fun ref( init : a ) : st⟨h⟩ ref⟨h,a⟩
```

```
fun (!)( r : ref⟨h,a⟩ ) : st⟨h⟩ a
{
  val x = r->value
  dup(x)
  x
}
```

dup a freed object!

```
fun (:=)( r : ref⟨h,a⟩, x : a ) : st⟨h⟩ ()
{
  val y = r->value
  r->value := x
  drop(y)
}
```

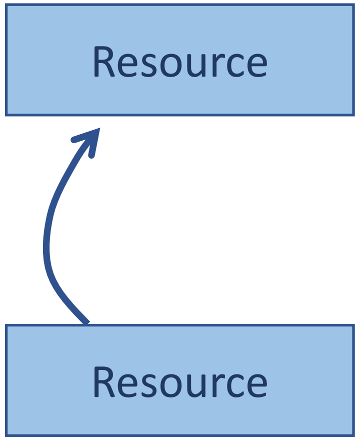
- **FBIP**: Functional but in-place
- **Thread-shared?** to avoid the atomic code path almost all the time.

3

# Mutation and cycles

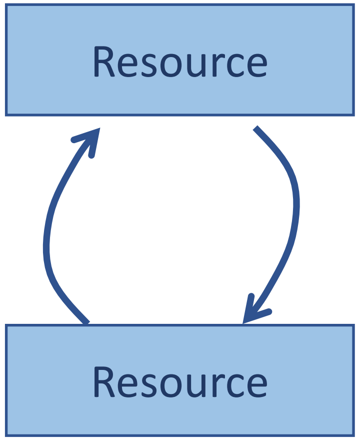
Resource

# Mutation and cycles

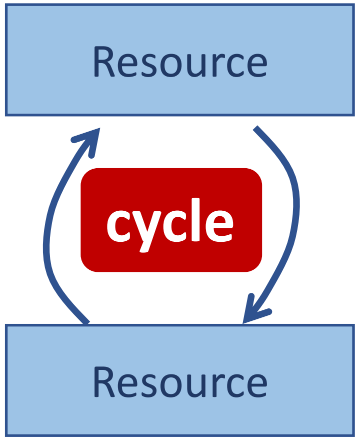


3

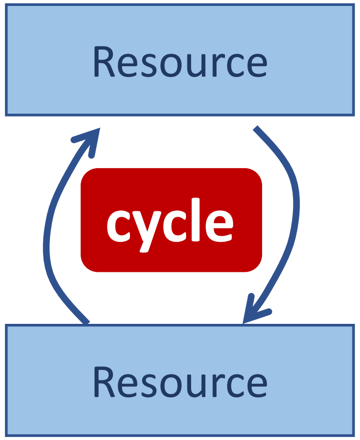
# Mutation and cycles



# Mutation and cycles

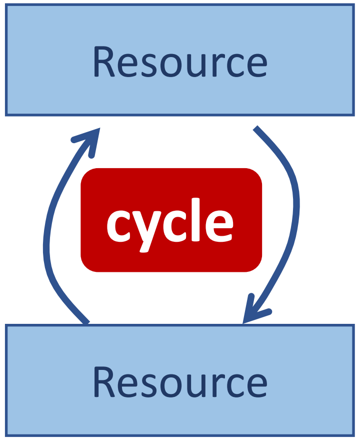


# Mutation and cycles



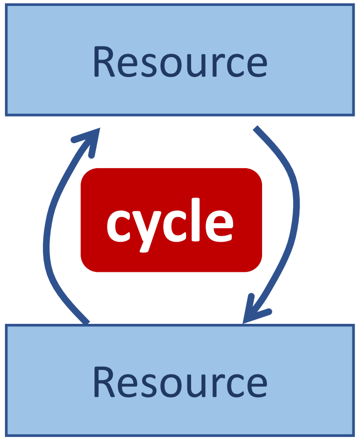

We leave the responsibility to the programmer to break cycles

# Mutation and cycles




We leave the responsibility to the programmer to break cycles (Swift)

# Mutation and cycles




We leave the responsibility to the programmer to break cycles (Swift)

Future improvements: generate code that tracks mutable data types at run time

# Koka references

- Koka: https://koka-lang.github.io/
- *Type Directed Compilation of Row-Typed Algebraic Effects.* Daan Leijen, POPL'17
- *Effect Handlers, Evidently.* Ningning Xie, Jonathan Brachthäuser, Daniel Hillerström, Philipp Schuster, Daan Leijen, ICFP'20
- *Generalized Evidence Passing for Effect Handlers.* Ningning Xie, Daan Leijen, under submission, Technical report MSR-TR-2021-5

# Agenda

① Perceus

② Koka 101

③ Functional But In-Place (FBIP)

④ Linear Resource Calculus

$\lambda^1$

# Agenda

① Perceus

② Koka 101

③ Functional But In-Place (FBIP)

④ Linear Resource Calculus

$\lambda^1$

# Reuse specialization

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion

# Reuse specialization

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru( dup(f) (x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

specialize

1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion

# Reuse specialization

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      Cons@ru (dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

specialize

1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion
4. reuse specialization

# Reuse specialization

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs);  Null
      Cons@ru( dup(f) (x), map(xx, f))
    }
    Nil {
      drop(xs);  drop(f);
      Nil
    }
  }
}
```

specialize

1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion
4. reuse specialization

```
fun Cons@ru( x, xx) {
  if (ru != NULL) {
  then {
     ru -> head := x;
     ru -> tail := xs;
     ru
  }
  else Cons(x, xx)
}
```

# Reuse specialization

```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs); Null
      if (ru != NULL) {
      then {
         ru -> head := x;
         ru -> tail := xs;
         ru
      }
      else Cons(x, xx)
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion
4. reuse specialization

```
fun Cons@ru( x, xx) {
  if (ru != NULL) {
  then {
     ru -> head := x;
     ru -> tail := xs;
     ru
  }
  else Cons(x, xx)
}
```

# Reuse specialization

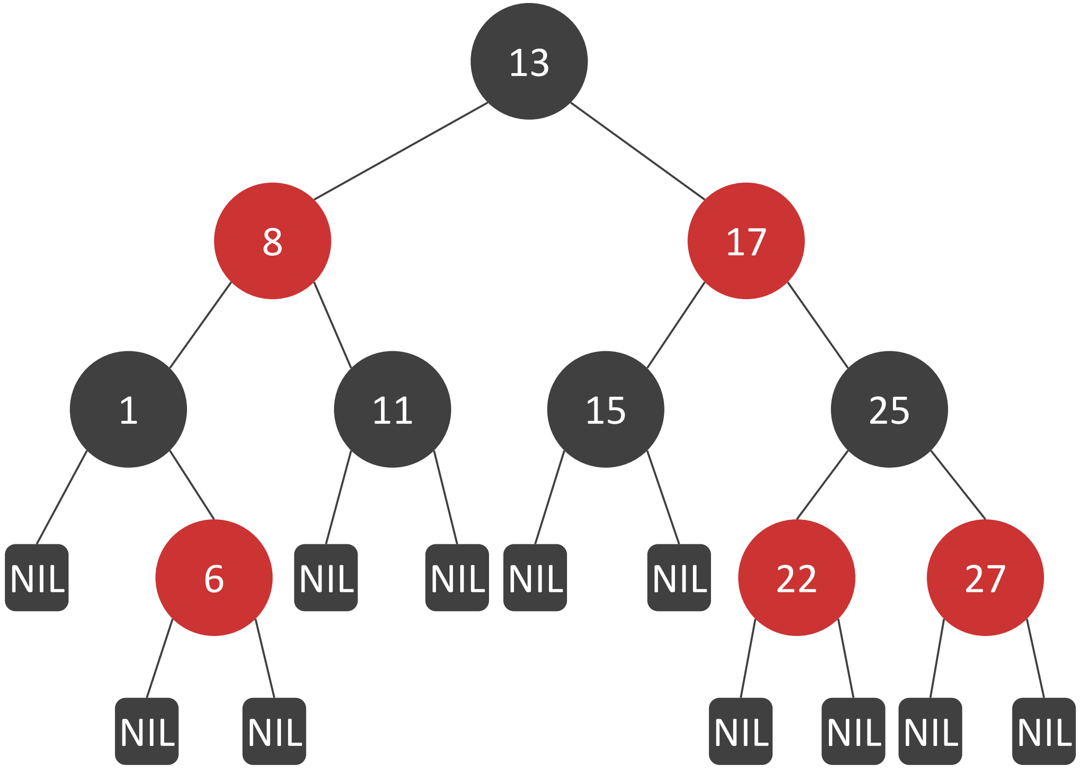
```
fun map(xs : list⟨a⟩, f : a -> b) : list⟨b⟩ {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
                  then &xs;
                  else dup(x); dup(xx);
                       decref(xs);  Null
      if (ru != NULL) {
      then {
         ru -> head := x;
         ru -> tail := xs;
         ru
      }
      else Cons(x, xx)
    }
    Nil {
      drop(xs); drop(f);
      Nil
    }
  }
}
```

For partial updates, we can further reuse unchanged fields of a construct

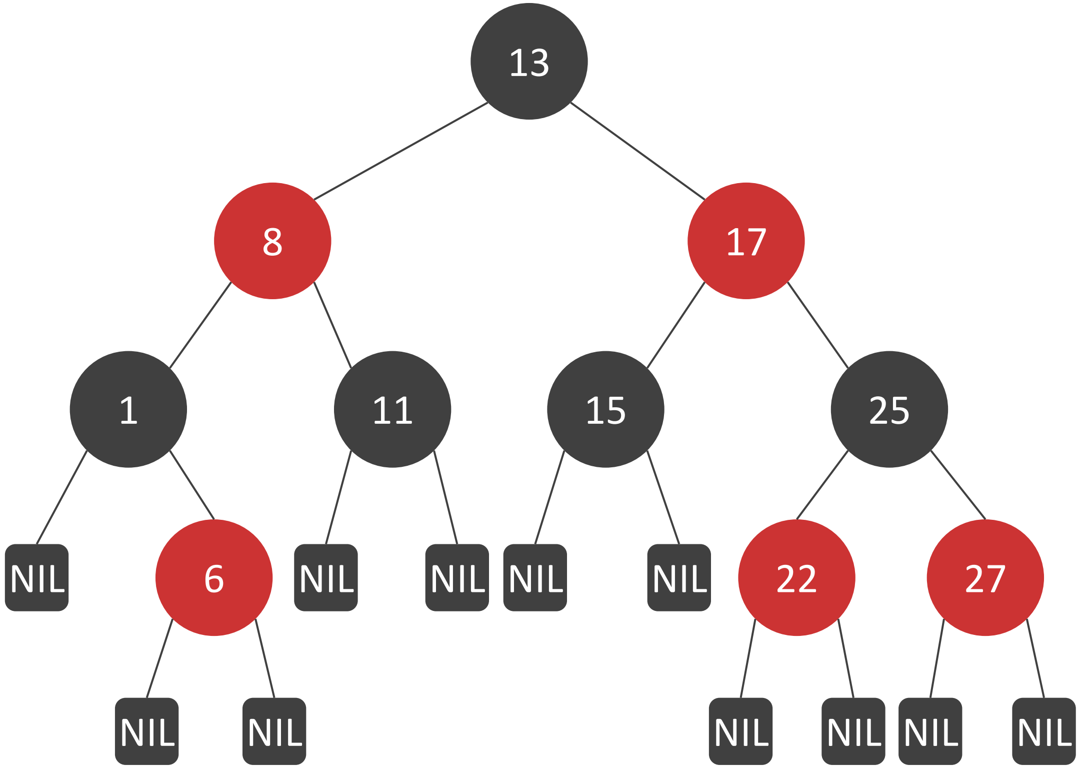
1. dup/drop insertion /reuse analysis
2. drop-reuse specialization
3. push down dup and fusion
4. reuse specialization

```
fun Cons@ru( x, xx) {
  if (ru != NULL) {
  then {
     ru -> head := x;
     ru -> tail := xs;
     ru
  }
  else Cons(x, xx)
}
```

# Red-black tree



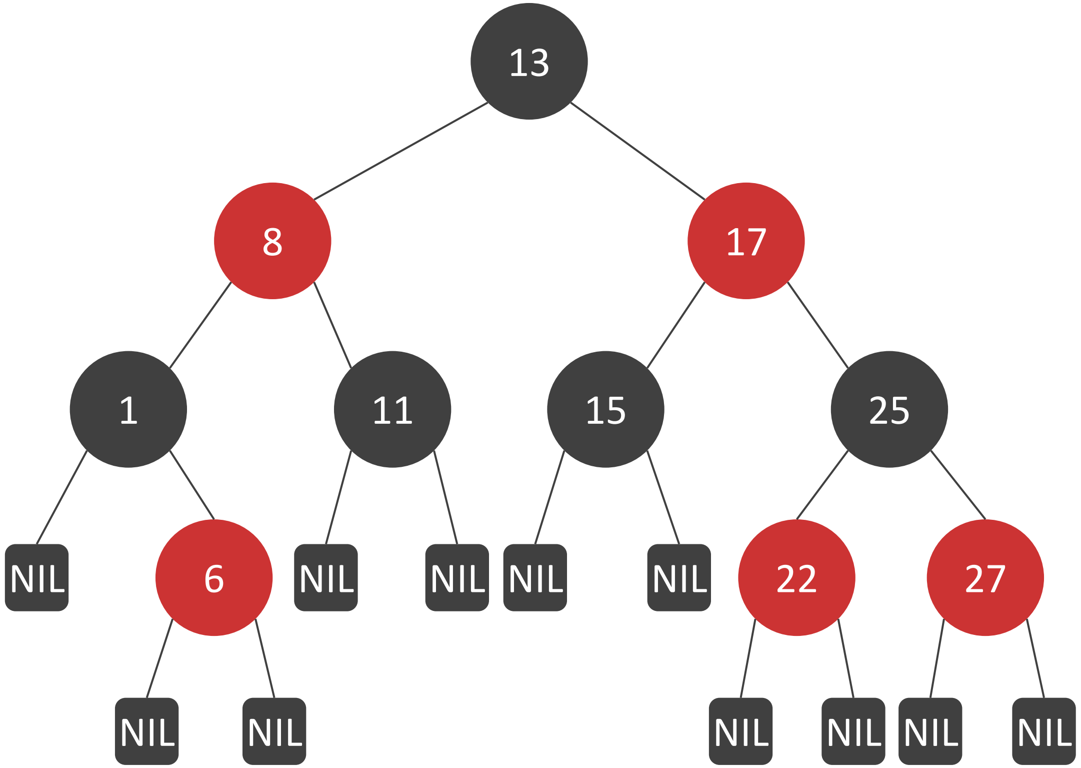
# Red-black tree




Each node is either **red** or **black**

- The root is **black**
- All leaves are **black**
- If a node is **red**, then its children are **black**

# Red-black tree

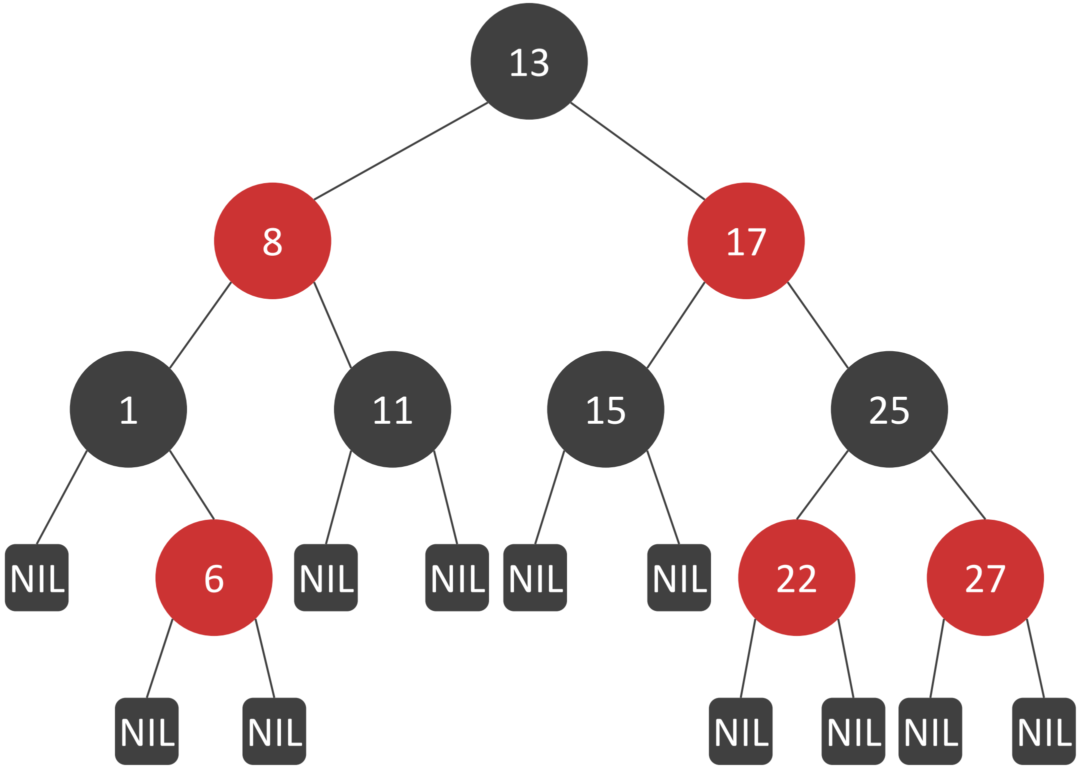



Each node is either **red** or **black**

- The root is **black**
- All leaves are **black**
- If a node is **red**, then its children are **black**

Every path from the root to any of the NIL leaves goes through the same number of **black** nodes.

# Red-black tree




Each node is either **red** or **black**

- The root is **black**
- All leaves are **black**
- If a node is **red**, then its children are **black**

Every path from the root to any of the NIL leaves goes through the same number of **black** nodes.

Search, delete and insert in $\mathcal{O}(\log(n))$

*J. Functional Programming* **1** (1): 1–000, January 1993  

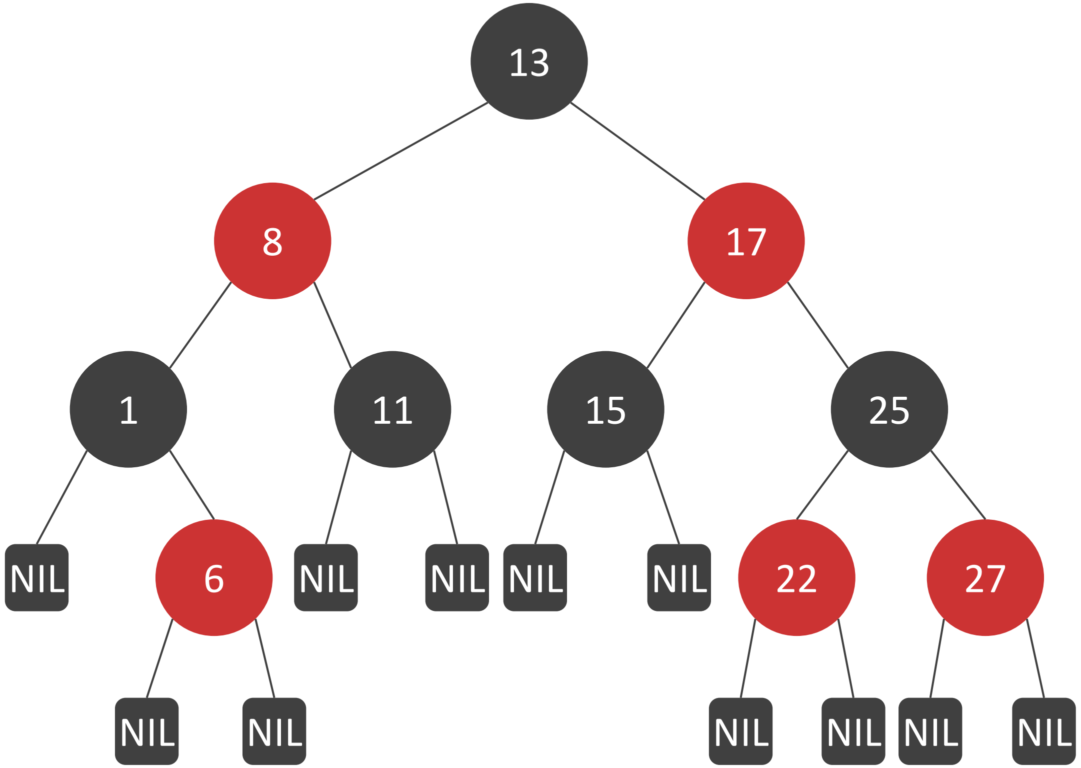
# FUNCTIONAL PEARLS

## *Red-Black Trees in a Functional Setting*

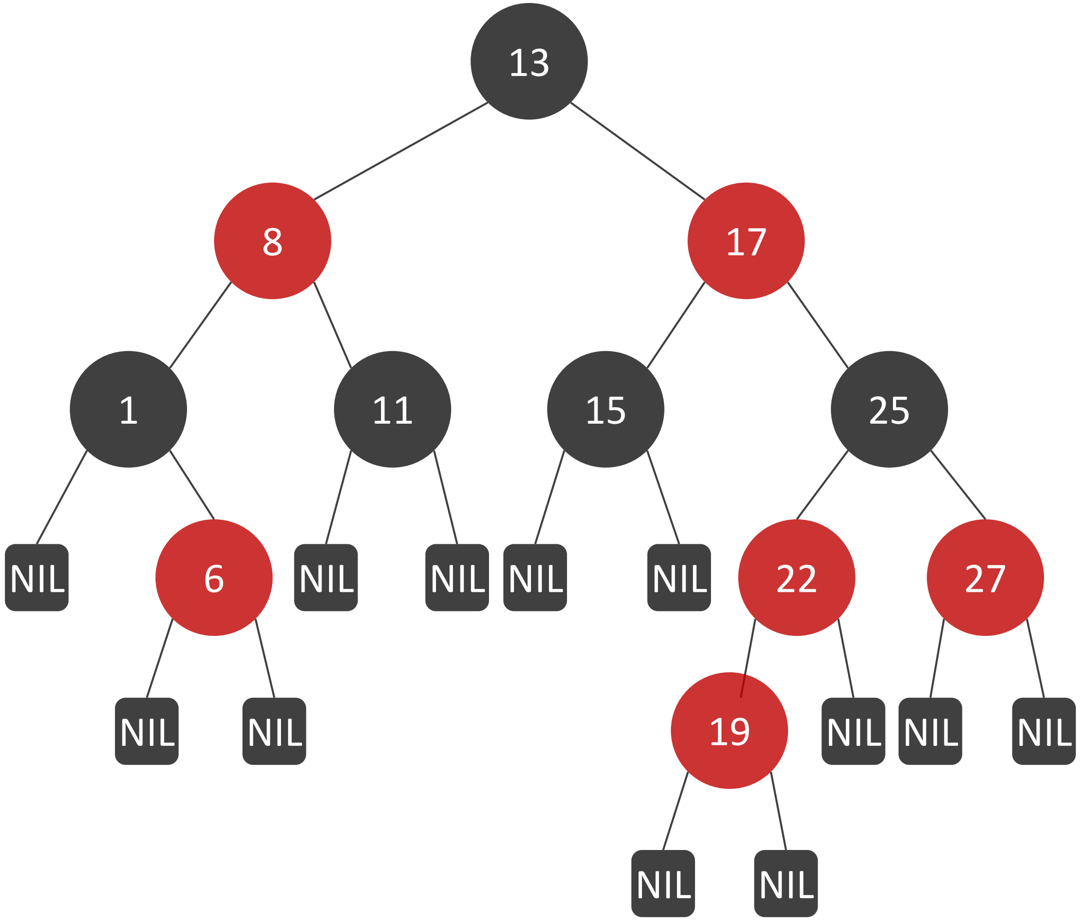
CHRIS OKASAKI†

*School of Computer Science, Carnegie Mellon University*
*5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA 15213*
*(e-mail:* cokasaki@cs.cmu.edu*)*

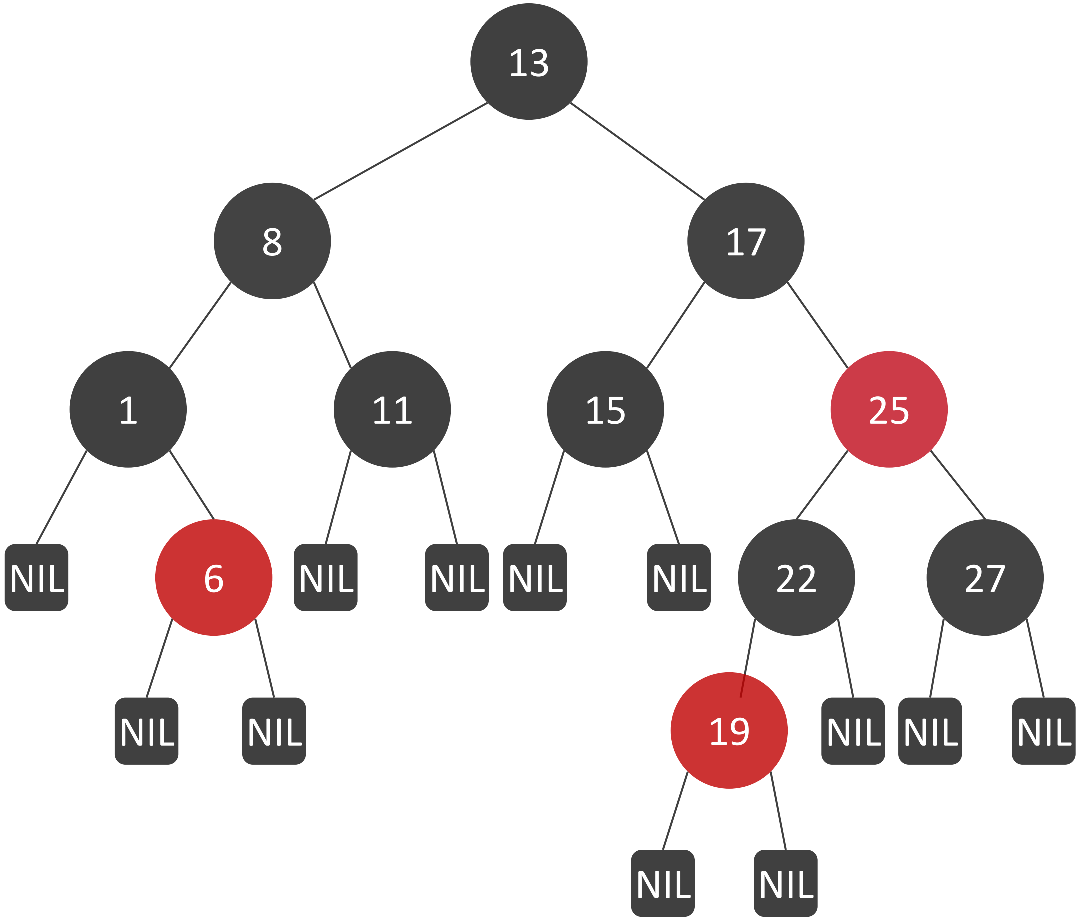
# Red-black tree insertion



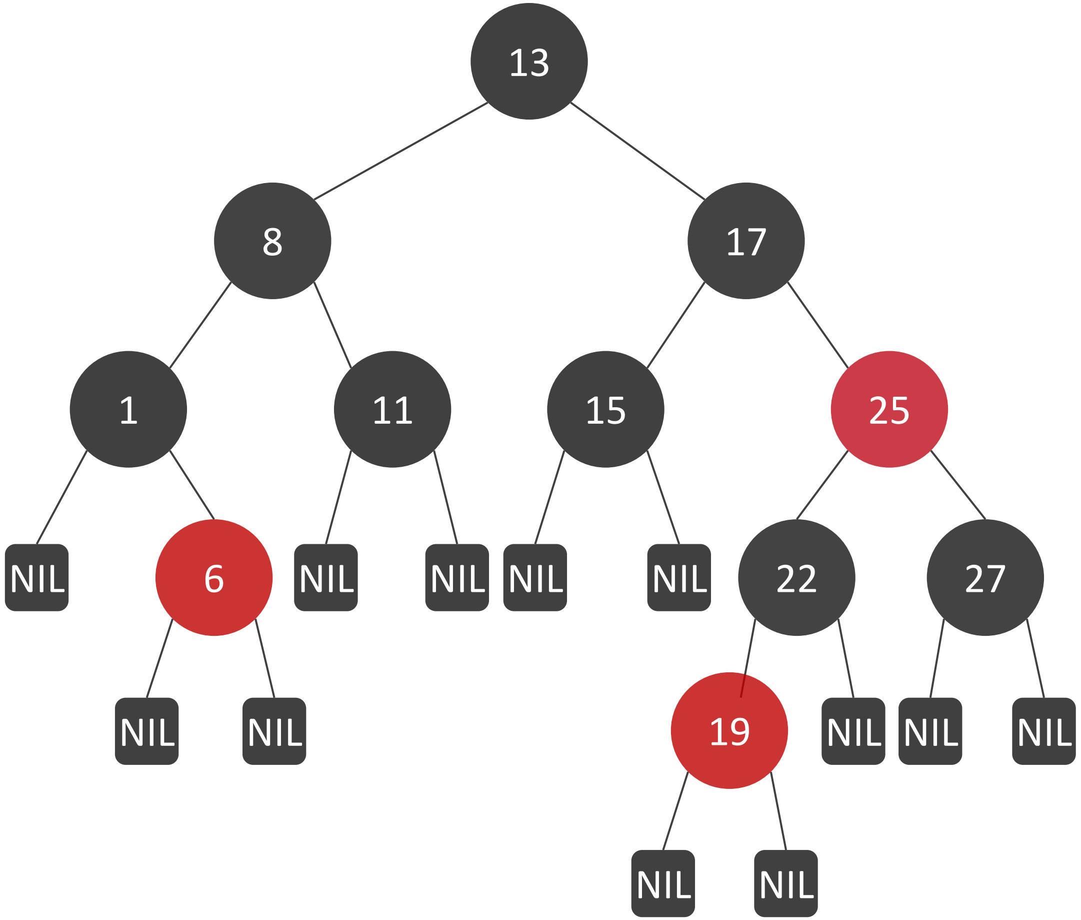
# Red-black tree insertion



# Red-black tree insertion

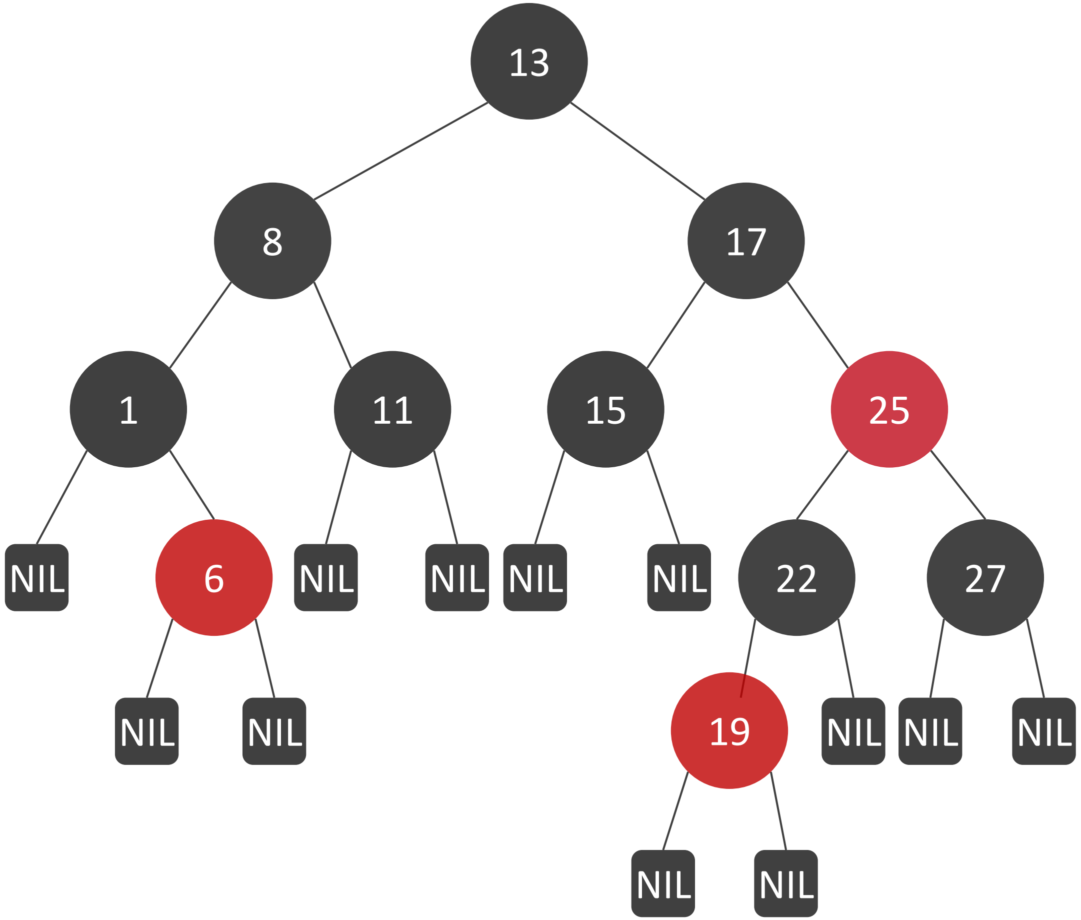


# Red-black tree insertion



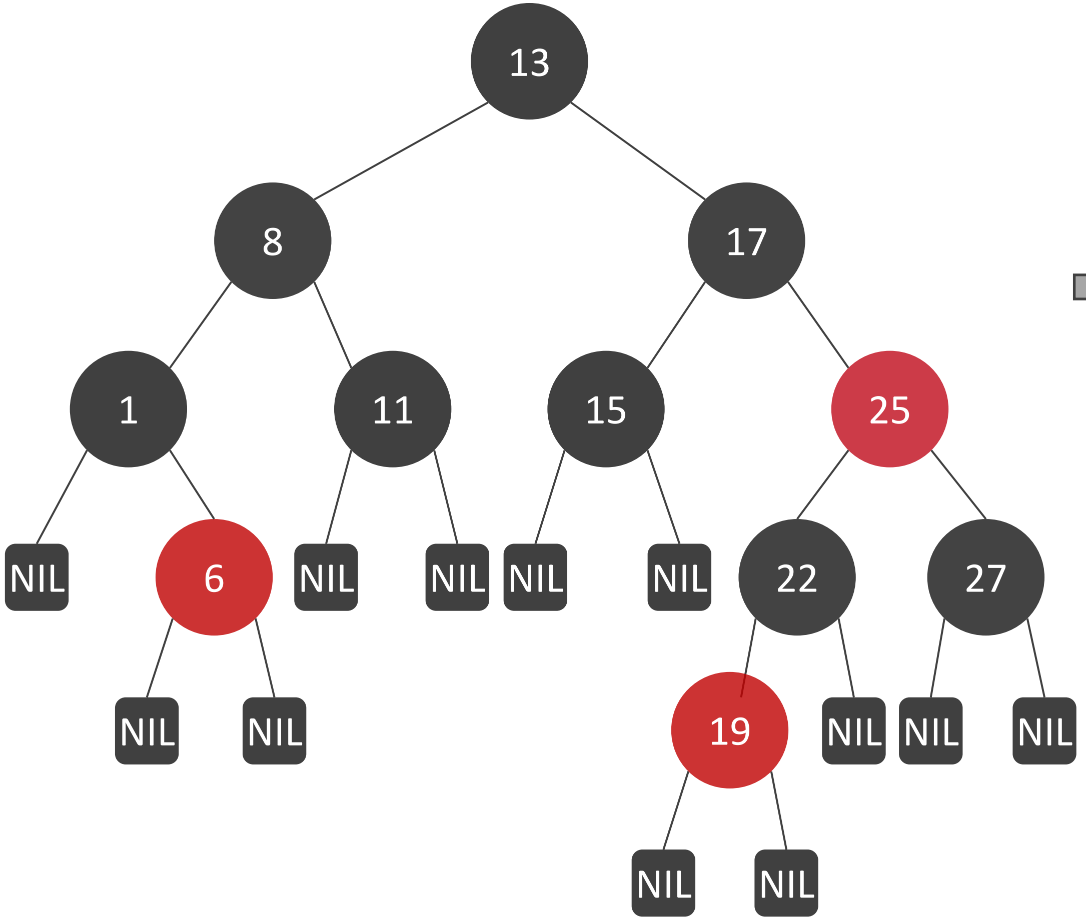

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

# Red-black tree insertion



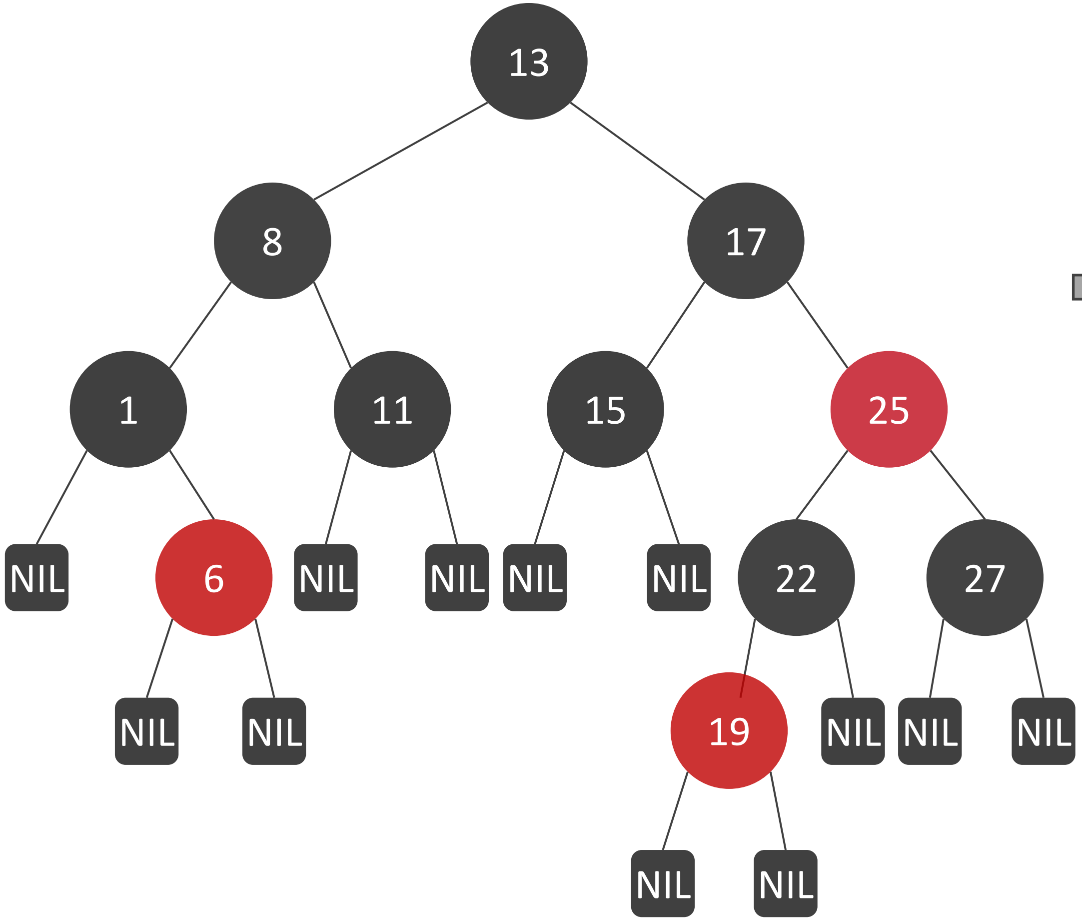

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

# Red-black tree insertion



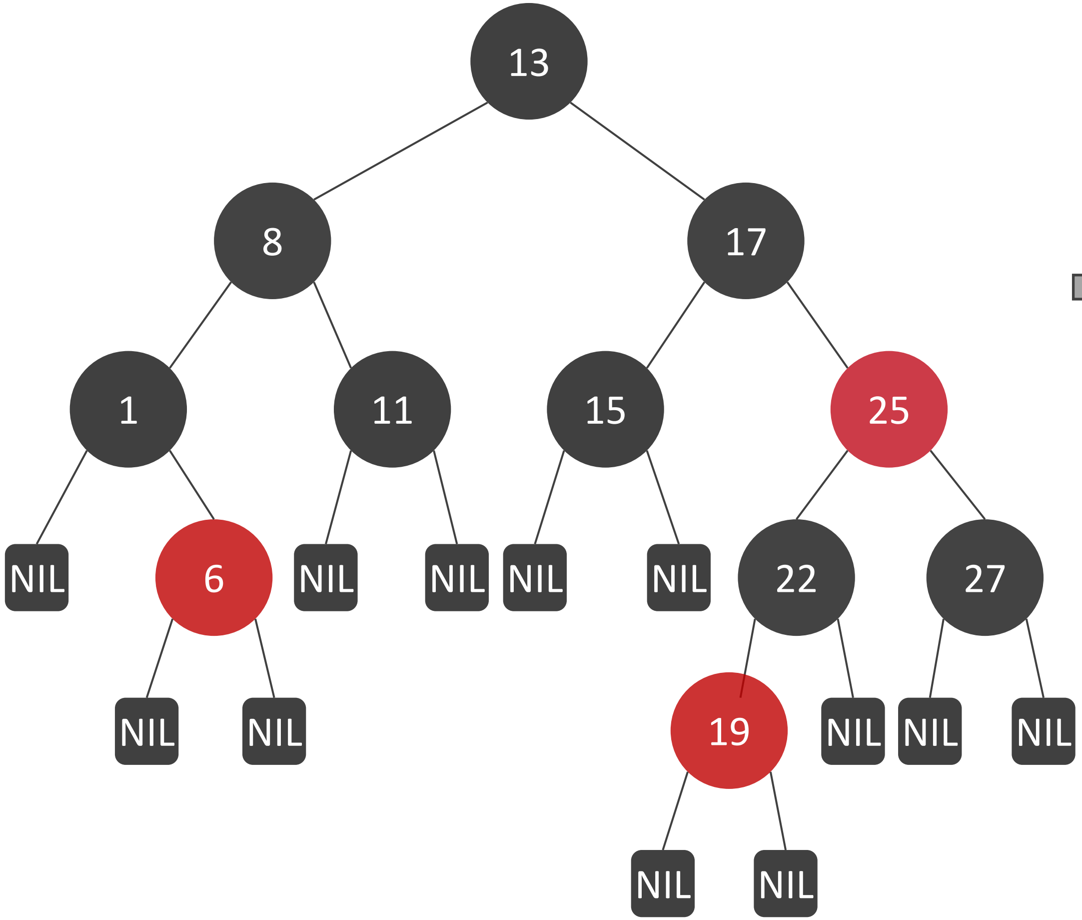

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

# Red-black tree insertion



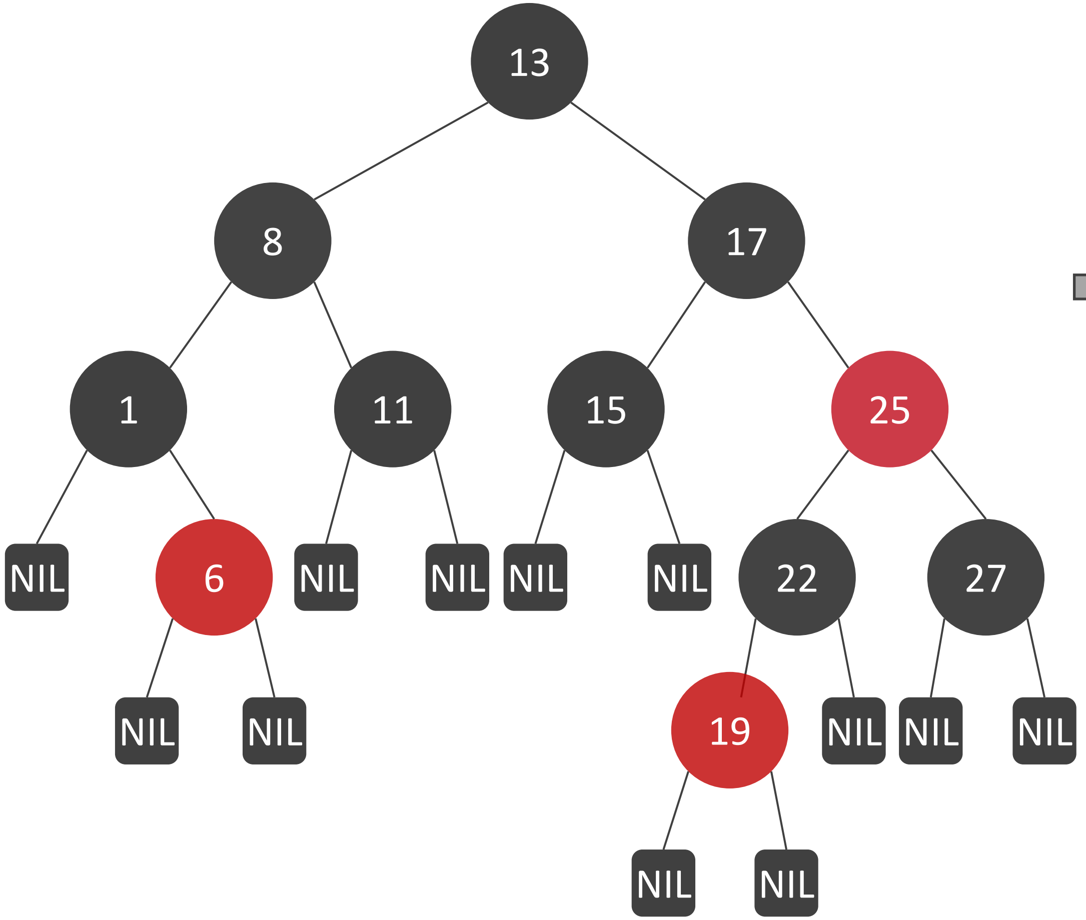

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

# Red-black tree insertion



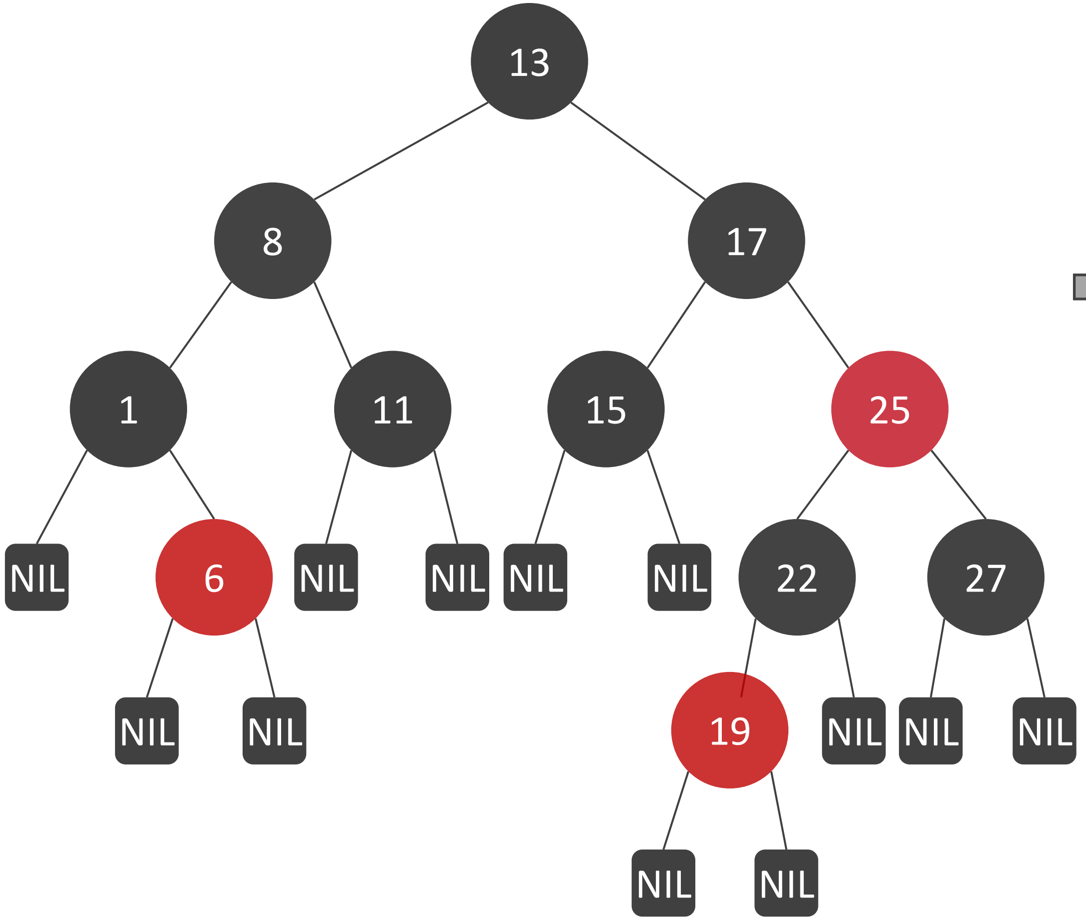

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

# Red-black tree insertion



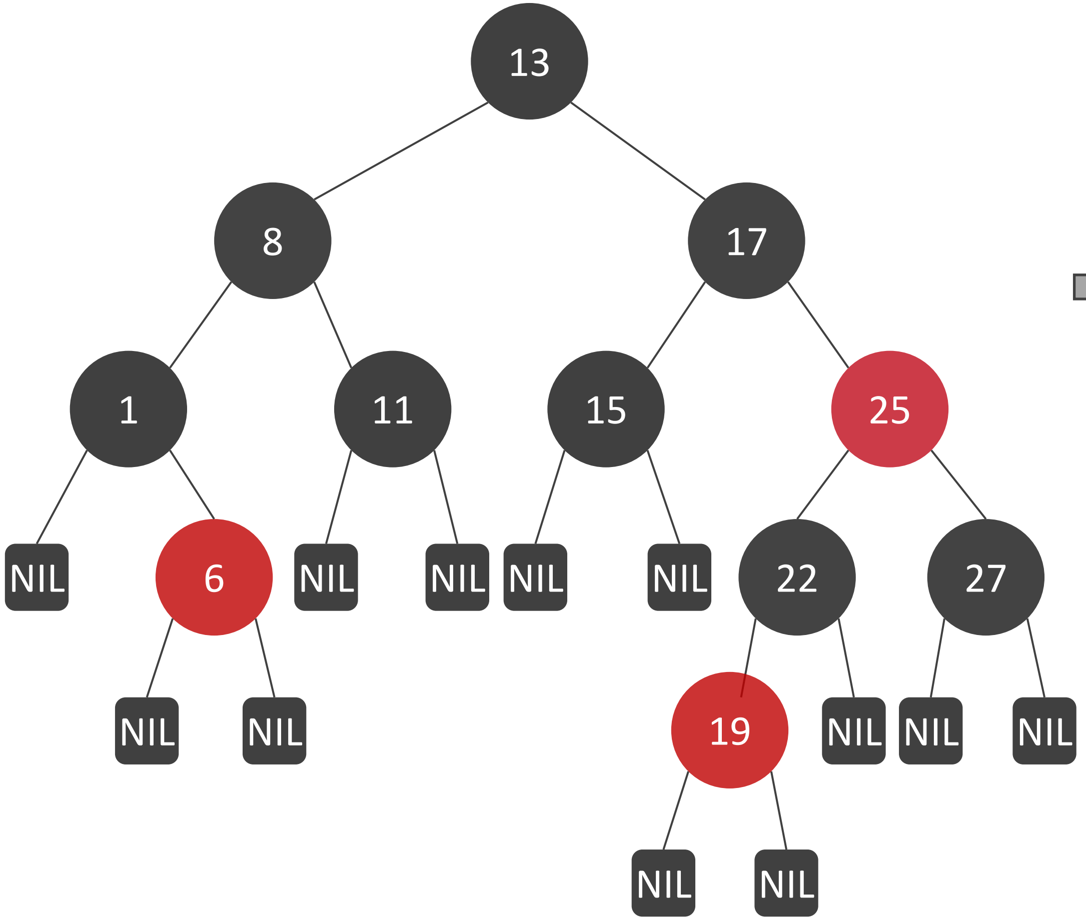

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

# Red-black tree insertion




```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

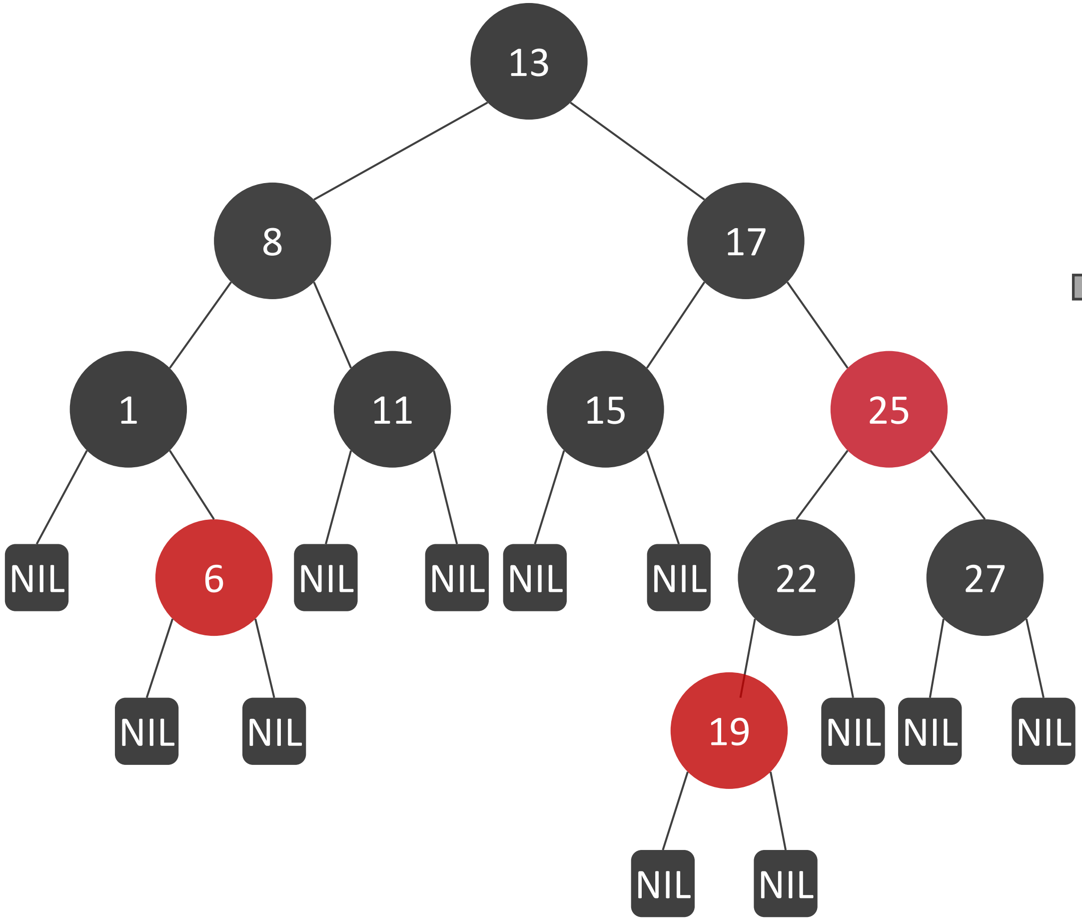
# Red-black tree insertion




```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      if (k < kx)
      then Node(Red, ins(l, k, v), kx, vx, r)
      elif (k == kx) then Node(Red, l, k, v, r)
      else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r) ->
        if (k < kx && is-red(l))
        then bal-left(ins(l,k,v), kx, vx, r)
        ...
  }
```

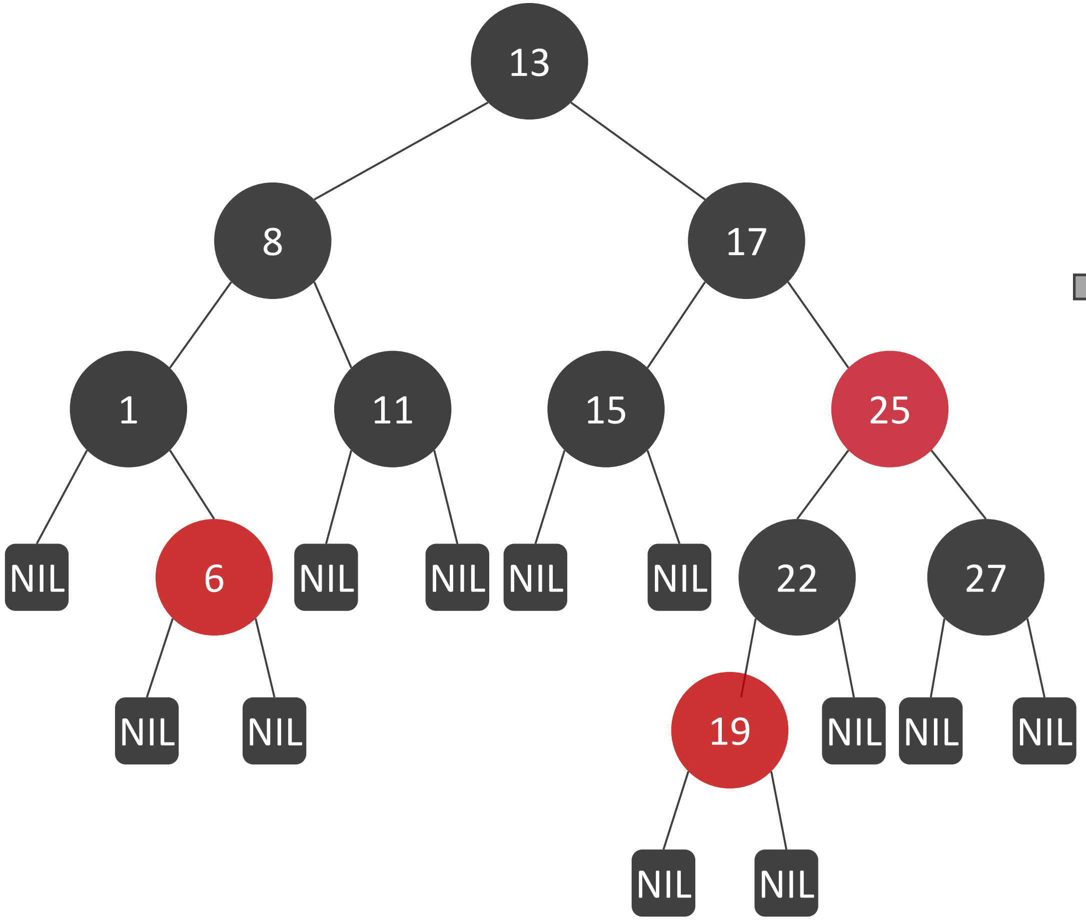
reuse analysis

# Red-black tree insertion




```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      val ru = if (is-unique(t)) then &t
               else { dup(l); dup(kx);
                      dup(vx); dup(r); NULL }
      if (dup(k) < dup(kx)) {
        Node @ru (Red, ins(l, k, v), kx, vx, r)
      }
```

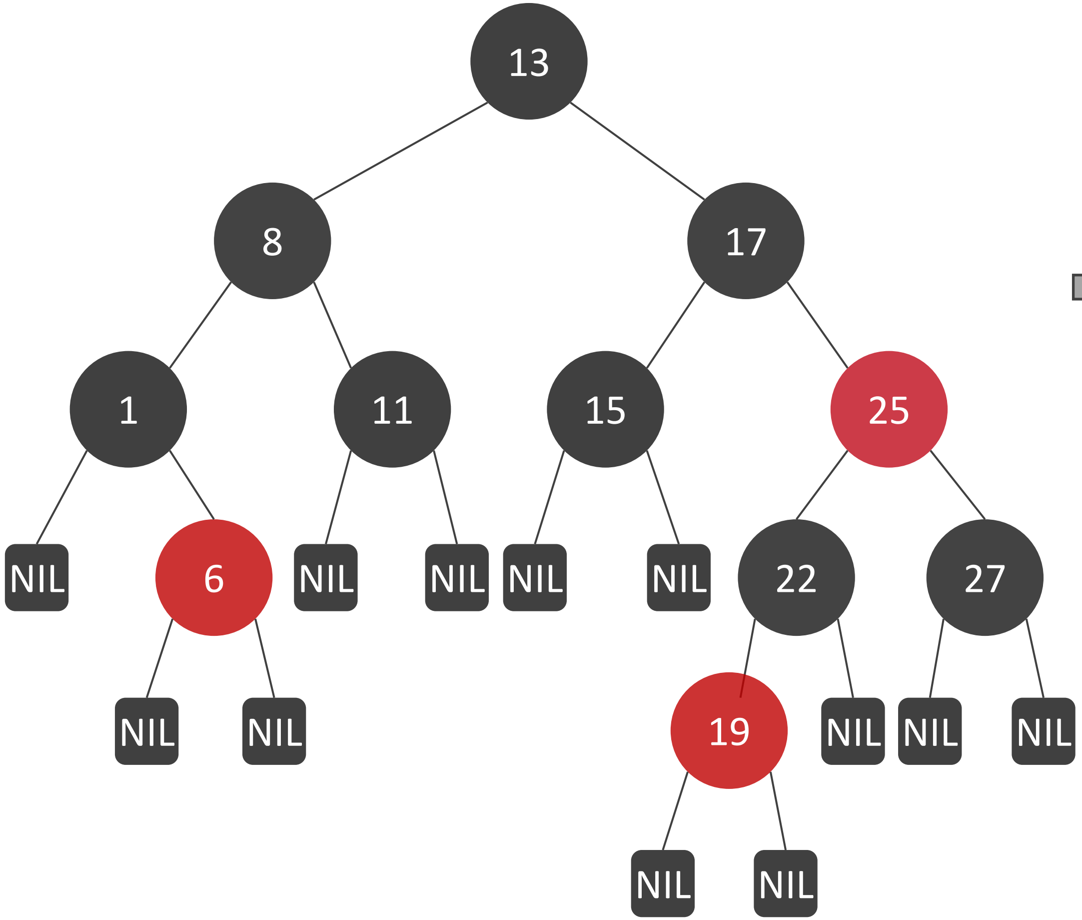
# Red-black tree insertion




```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      val ru = if (is-unique(t)) then &t
               else { dup(l); dup(kx);
                      dup(vx); dup(r); NULL }
      if (dup(k) < dup(kx)) {
        Node @ru (Red, ins(l, k, v), kx, vx, r)
      }
```

partial update

# Red-black tree insertion

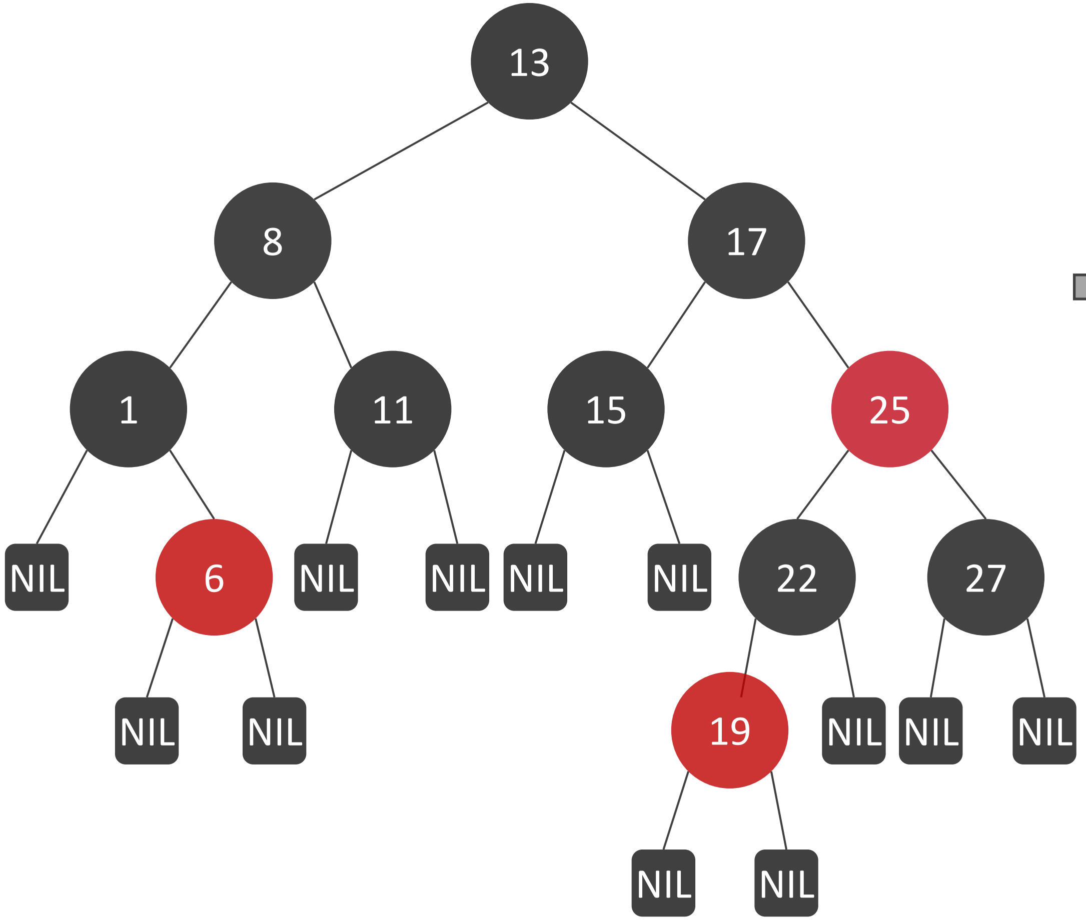



```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      val ru = if (is-unique(t)) then &t
               else { dup(l); dup(kx);
                      dup(vx); dup(r); NULL }
      if (dup(k) < dup(kx)) {
        Node @ru (Red, ins(l, k, v), kx, vx, r)
      }
```

reuse specialize

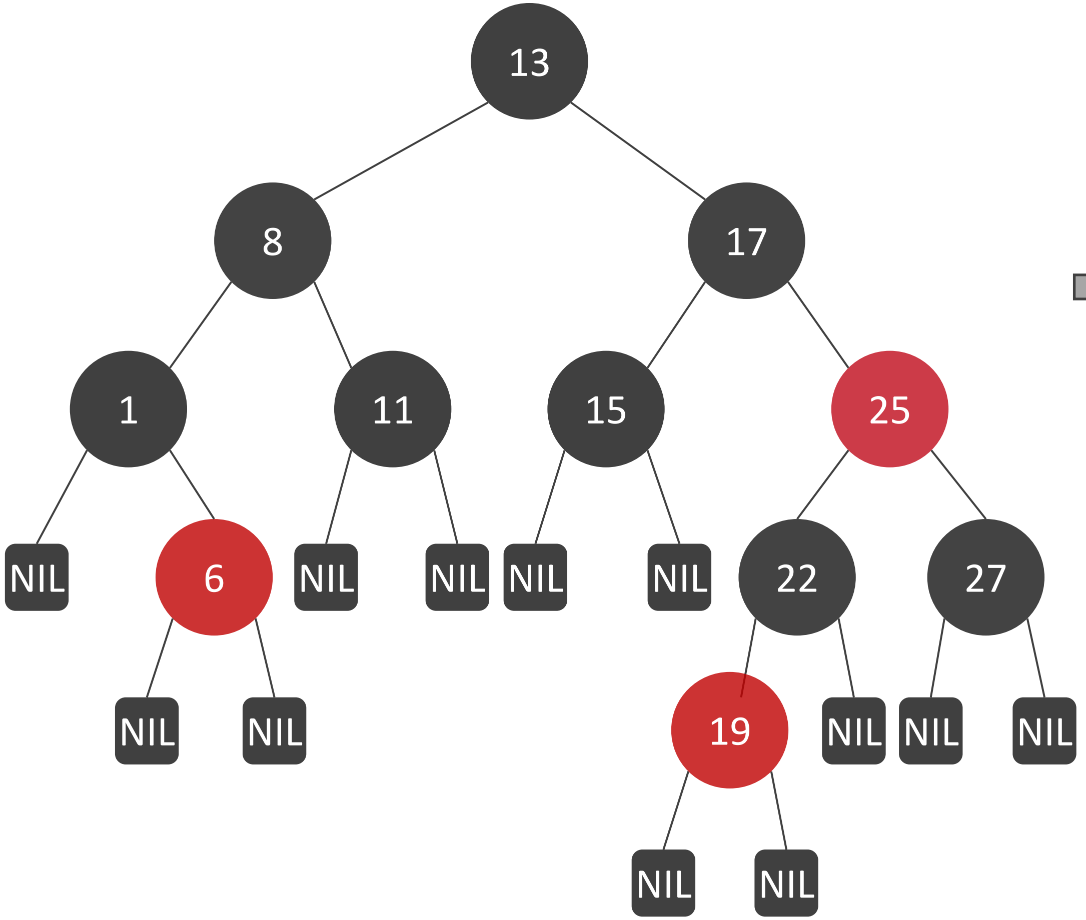
partial update

# Red-black tree insertion




```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      val ru = if (is-unique(t)) then &t
               else { dup(l); dup(kx);
                      dup(vx); dup(r); NULL }

        val y = ins(l, k ,v)
        if (ru != NULL)
        then { ru ->left := y;
               ru
        }
        else Node(Red, y, kx, vx, r)
      }
```

# Red-black tree insertion



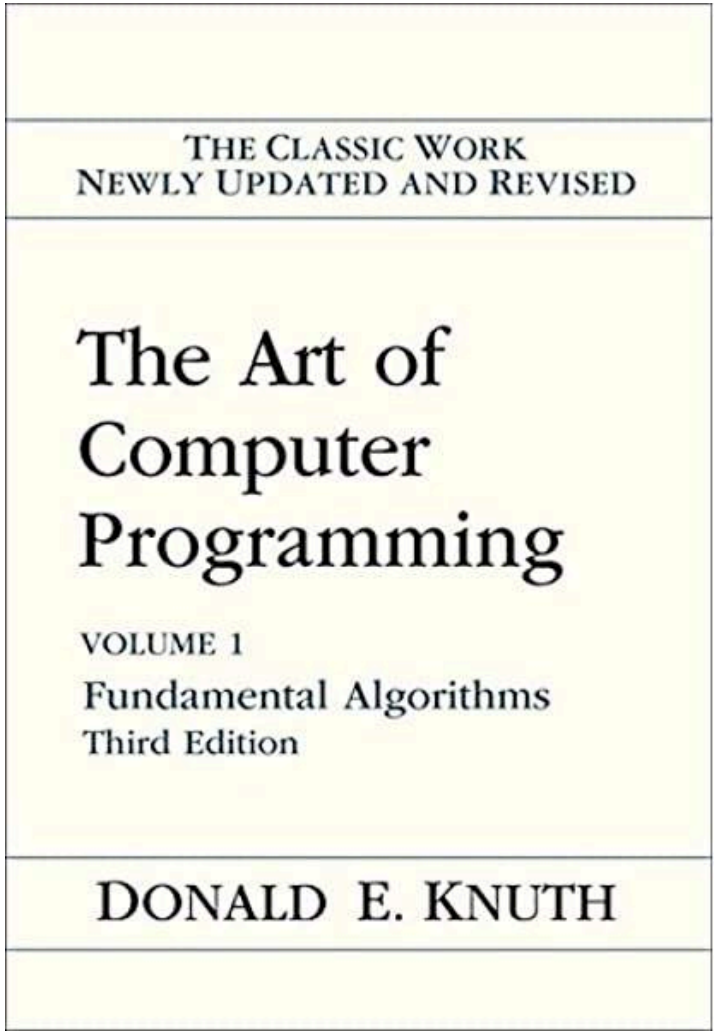

```
fun ins( t : tree, k : int, v : bool ): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) ->
      val ru = if (is-unique(t)) then &t
               else { dup(l); dup(kx);
                      dup(vx); dup(r); NULL }

        val y = ins(l, k ,v)
        if (ru != NULL)
        then { ru ->left := y;
               ru
        }
        else Node(Red, y, kx, vx, r)
  }
```

reuse unchanged fields of a construct

# FBIP: Functional but in-place

For a unique resource, the purely functional algorithm adapts at runtime to an in-place mutating algorithm

# FBIP Application




Challenge: visiting a tree in-order while using no extra stack- or heap space

# Morris in-order tree traversal algorithm in C

```
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

# Morris in-order tree traversal algorithm in C

```
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Initialize the root as the current node

# Morris in-order tree traversal algorithm in C

```
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Initialize the root as the current node

apply f
visit right tree

# Morris in-order tree traversal algorithm in C

```c
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Initialize the root as the current node

apply f
visit right tree

make cursor the right child of the
rightmost node in cursor's left subtree

# Morris in-order tree traversal algorithm in C

```
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Initialize the root as the current node

apply f
visit right tree

make cursor the right child of the rightmost node in cursor's left subtree

visit left tree

# Morris in-order tree traversal algorithm in C

```c
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Initialize the root as the current node

apply f
visit right tree

make cursor the right child of the rightmost node in cursor's left subtree

visit left tree

apply f
visit right tree

# Morris in-order tree traversal algorithm in C

```c
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Initialize the root as the current node

apply f
visit right tree

make cursor the right child of the rightmost node in cursor's left subtree

visit left tree

apply f
visit right tree

in-place mutating algorithm that swaps pointers in the tree to "remember" which parts are unvisited.

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}

type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                             // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}
type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                             // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

an explicit visitor data structure that keeps track of which parts of the tree we still need to visit.

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}
type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                             // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

an explicit visitor data structure that keeps track of which parts of the tree we still need to visit.

a direction data structure

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}

type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                             // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

an explicit visitor data structure that keeps track of which parts of the tree we still need to visit.

a direction data structure

pattern match on directions, trees, and visitors

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}

type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done         -> t                            // C
      BinR(r,x,v)  -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v)  -> tmap(f,Bin(l,x,t),v,Up)      // E
} } }
```

reuse analysis

an explicit visitor data structure that keeps track of which parts of the tree we still need to visit.

a direction data structure

pattern match on directions, trees, and visitors

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}

type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                              // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

reuse analysis

tail call

an explicit visitor data structure that keeps track of which parts of the tree we still need to visit.

a direction data structure

pattern match on directions, trees, and visitors

# FBIP in-order tree traversal algorithm in Koka

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}

type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                             // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

reuse analysis

tail call

an explicit visitor data structure that keeps track of which parts of the tree we still need to visit.

a direction data structure

a purely functional specification with in-place updating

pattern match on directions, trees, and visitors

# FBIP

```c
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    if (cursor->left == NULL) {
      // no left tree, go down the right
      f(cursor->value);
      cursor = cursor->right;
    } else {
      // has a left tree
      tree* pre = cursor->left;  // find the predecessor
      while(pre->right != NULL && pre->right != cursor) {
        pre = pre->right;
      }
      if (pre->right == NULL) {
        // first visit, remember to visit right tree
        pre->right = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}
type visitor {
  Done
  BinR( right:tree, value : int, visit : visitor )
  BinL( left:tree, value : int, visit : visitor )
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree,
          visit : visitor, d : direction ) : tree {
  match(d) {
    Down -> match(t) {    // going down a left spine
      Bin(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done        -> t                             // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

# Agenda

① Perceus

② Koka 101

③ Functional But In-Place (FBIP)

④ Linear Resource Calculus

$\lambda^1$

# Agenda

① Perceus

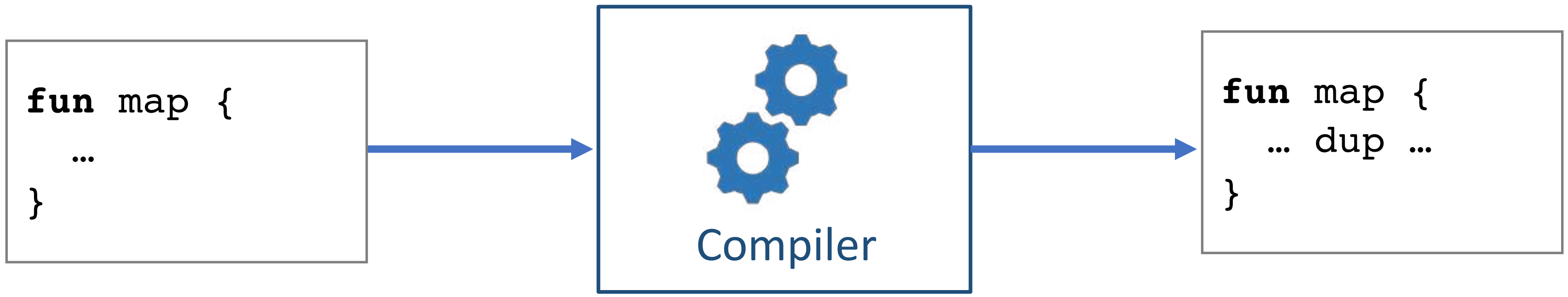
② Koka 101

③ Functional But In-Place (FBIP)

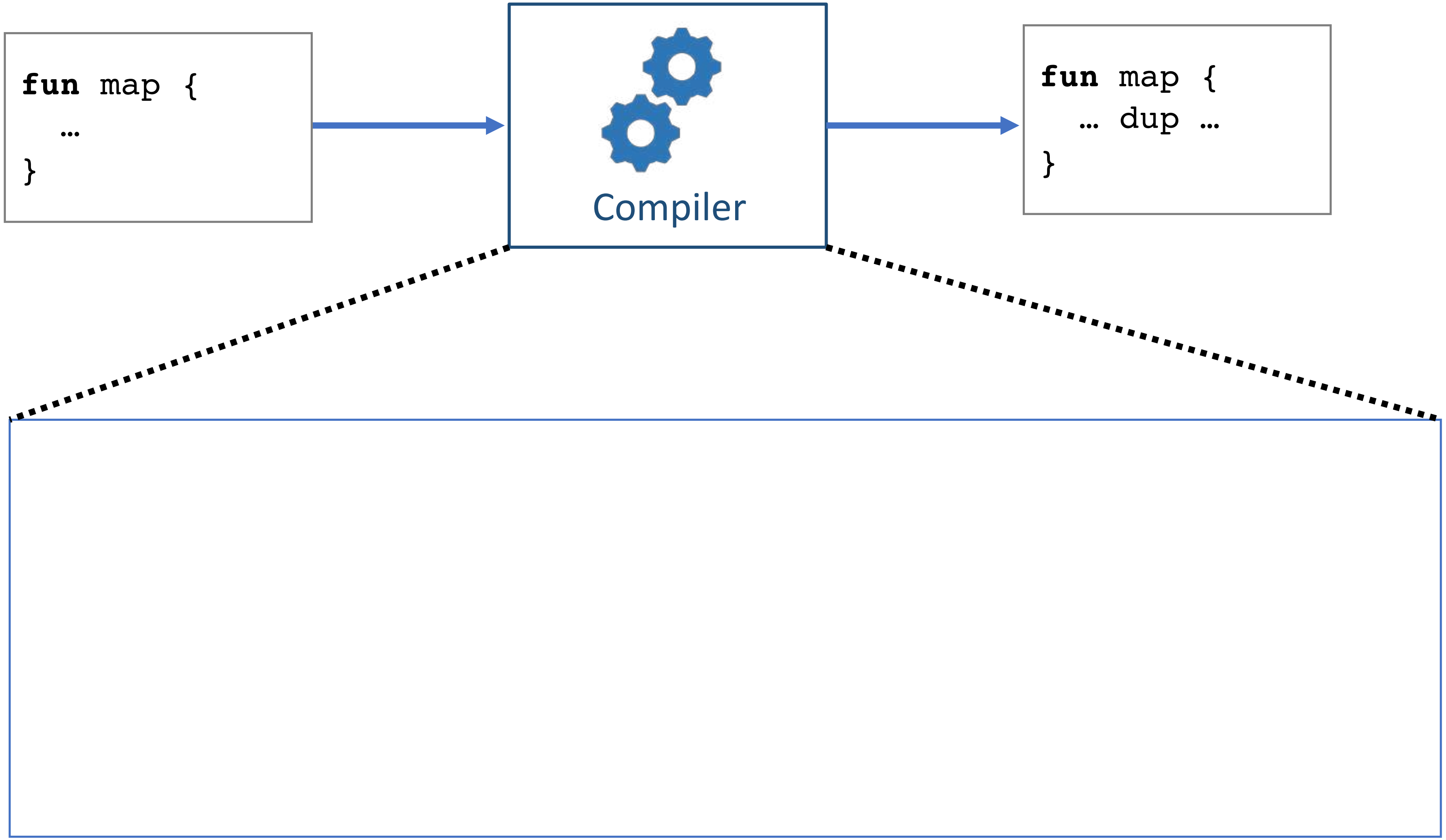
④ Linear Resource Calculus

$\lambda^1$

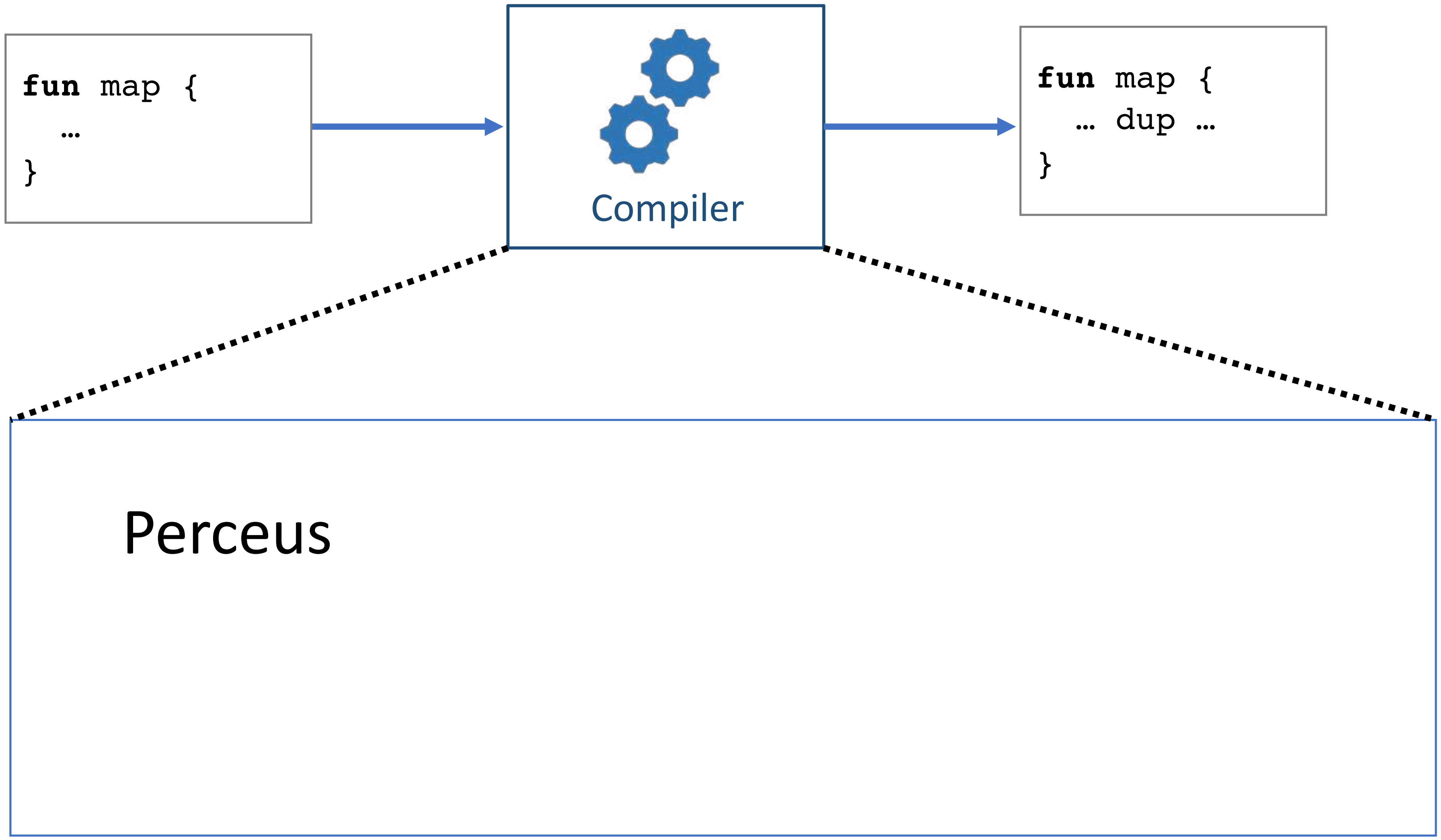
# Architecture



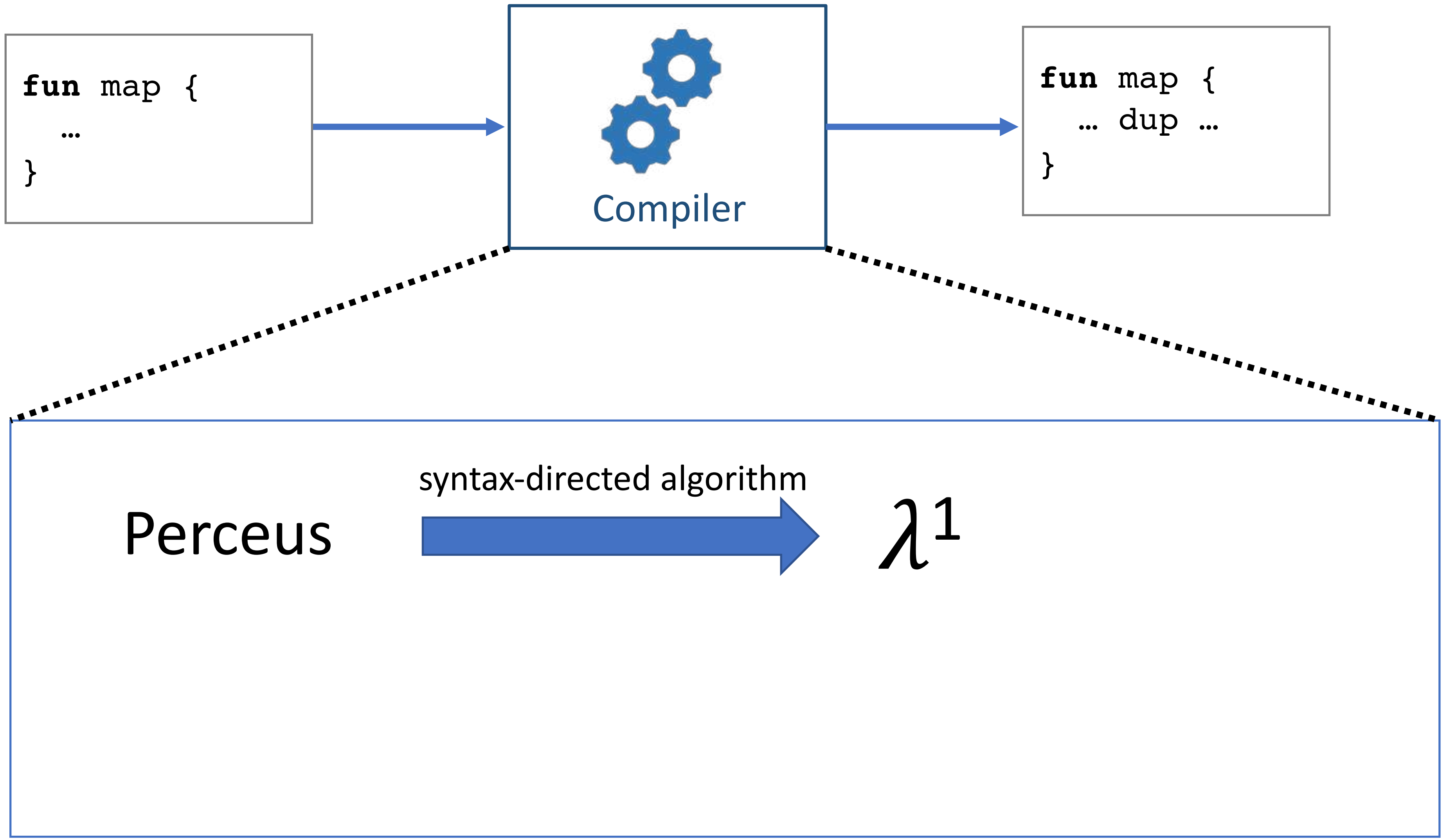
# Architecture



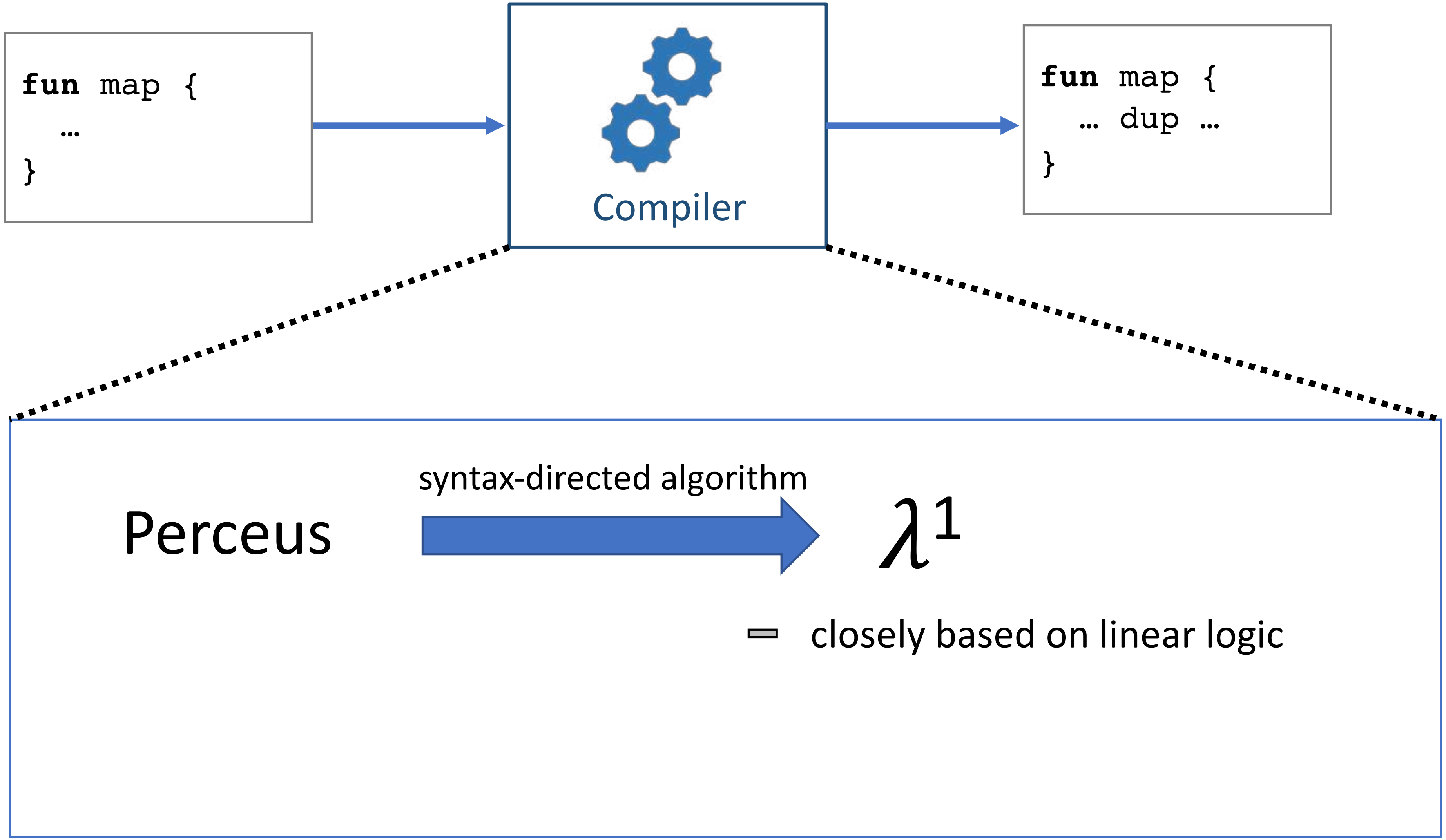
# Architecture



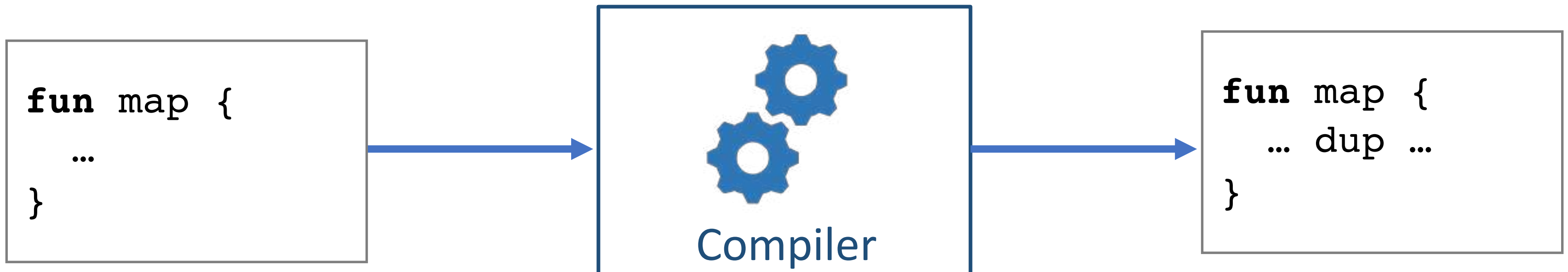
# Architecture



# Architecture



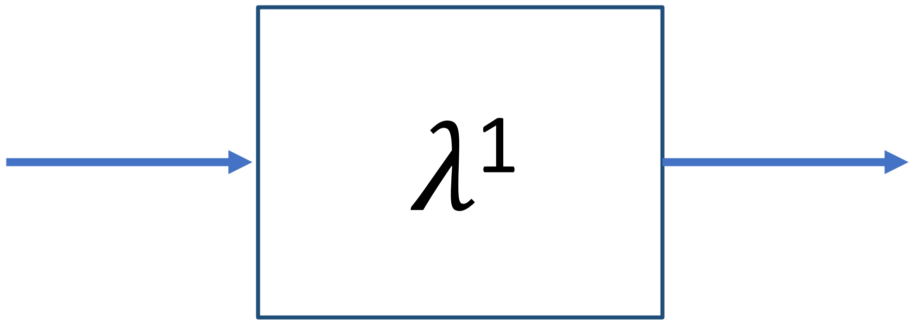
# Architecture




Perceus $\xrightarrow{\text{syntax-directed algorithm}}$ $\lambda^1$

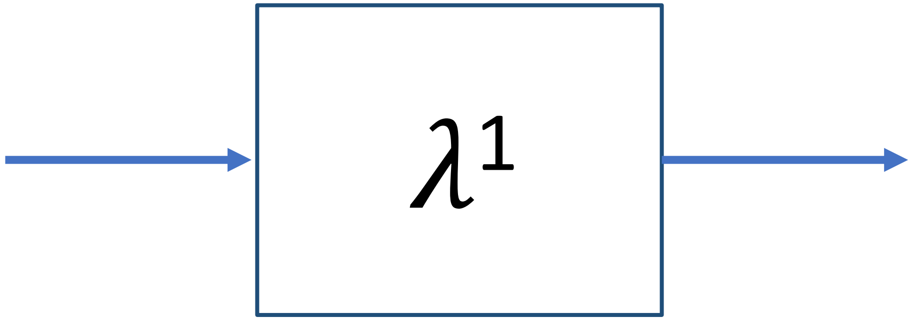
- closely based on linear logic
- operational semantics in an explicit heap with reference counting

# A linear resource calculus



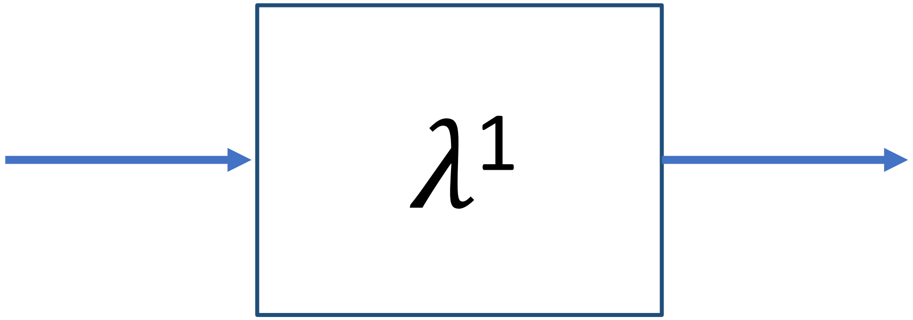
# A linear resource calculus

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
 & \mid & \textsf{val}\ x\ =\ e;\ e \\
 & \mid & \textsf{match}\ x\ \{\ \overline{p_i \longrightarrow e_i}\ \}
\end{array}
$$



# A linear resource calculus

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$




$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
 & \mid & \mathsf{dup}\ x;\ e \\
 & \mid & \mathsf{drop}\ x;\ e \\
 & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

# A linear resource calculus

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
& \mid & \mathsf{val}\ x\ =\ e;\ e \\
& \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

No mutable references

$$\lambda^1$$

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
& \mid & \mathsf{val}\ x\ =\ e;\ e \\
& \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
& \mid & \mathsf{dup}\ x;\ e \\
& \mid & \mathsf{drop}\ x;\ e \\
& \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

# A linear resource calculus

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$$\longrightarrow \boxed{\lambda^1} \longrightarrow$$

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
 & \mid & \mathsf{dup}\ x;\ e \\
 & \mid & \mathsf{drop}\ x;\ e \\
 & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

# A linear resource calculus

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
& \mid & \mathsf{val}\ x\ =\ e;\ e \\
& \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$$\longrightarrow \boxed{\lambda^1} \longrightarrow$$

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
& \mid & \mathsf{val}\ x\ =\ e;\ e \\
& \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
& \mid & \mathsf{dup}\ x;\ e \\
& \mid & \mathsf{drop}\ x;\ e \\
& \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$$\Delta \mid \Gamma \vdash e \rightsquigarrow e'$$

# A linear resource calculus

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
& \mid & \mathsf{val}\ x\ =\ e;\ e \\
& \mid & \mathsf{match}\ x\ \{\ \overline{p_i \longrightarrow e_i}\ \}
\end{array}
$$

$\lambda^1$

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
& \mid & \mathsf{val}\ x\ =\ e;\ e \\
& \mid & \mathsf{match}\ x\ \{\ \overline{p_i \longrightarrow e_i}\ \} \\
& \mid & \mathsf{dup}\ x;\ e \\
& \mid & \mathsf{drop}\ x;\ e \\
& \mid & \mathsf{match}\ e\ \{\ \overline{p_i \longrightarrow e_i}\ \}
\end{array}
$$

**Context** $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

resources in scope

# A linear resource calculus

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
  & \mid & \mathsf{val}\ x\ =\ e;\ e \\
  & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$\lambda^1$

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
  & \mid & \mathsf{val}\ x\ =\ e;\ e \\
  & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
  & \mid & \mathsf{dup}\ x;\ e \\
  & \mid & \mathsf{drop}\ x;\ e \\
  & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

borrowed

**Context** $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

resources in scope

# A linear resource calculus

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
  & \mid & \mathsf{val}\ x\ =\ e;\ e \\
  & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$\lambda^1$

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \\
  & \mid & \mathsf{val}\ x\ =\ e;\ e \\
  & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
  & \mid & \mathsf{dup}\ x;\ e \\
  & \mid & \mathsf{drop}\ x;\ e \\
  & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

borrowed owned

**Context** $\Delta \mid \Gamma \vdash e \leadsto e'$

resources in scope

# A linear resource calculus

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
 & \mid & \text{val } x = e;\ e \\
 & \mid & \text{match } x \ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$\lambda^1$

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
 & \mid & \text{val } x = e;\ e \\
 & \mid & \text{match } x \ \{\ \overline{p_i \to e_i}\ \} \\
 & \mid & \text{dup } x;\ e \\
 & \mid & \text{drop } x;\ e \\
 & \mid & \text{match } e \ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

input

borrowed owned

**Context** $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

resources in scope

# A linear resource calculus

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

$\longrightarrow$ $\boxed{\lambda^1}$ $\longrightarrow$

$$
\begin{array}{lcl}
e & ::= & v \mid e\ e \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} \\
 & \mid & \mathsf{dup}\ x;\ e \\
 & \mid & \mathsf{drop}\ x;\ e \\
 & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
$$

input output

borrowed owned

**Context** $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

resources in scope

# Declarative linear resource rules

$$\text{Context} \quad \overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash \overset{\text{input}}{e} \rightsquigarrow \overset{\text{output}}{e'}$$

# Declarative linear resource rules

$$\text{Context}\ \overbrace{\Delta}^{\text{borrowed}} \mid \overbrace{\Gamma}^{\text{owned}} \vdash \overbrace{e}^{\text{input}} \leadsto \overbrace{e'}^{\text{output}}$$

$$\frac{}{\Delta \mid x \vdash x \leadsto x}\ [\text{VAR}]$$

# Declarative linear resource rules

$$\text{Context}\quad \overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash \overset{\text{input}}{e} \rightsquigarrow \overset{\text{output}}{e'}$$

$$\frac{}{\Delta \mid x \vdash \overset{\text{consume a resource}}{x} \rightsquigarrow x}\ [\textsc{Var}]$$

# Declarative linear resource rules

$$\text{Context}\quad \overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash \overset{\text{input}}{e} \rightsquigarrow \overset{\text{output}}{e'}$$

own and only own it exactly once

consume a resource

$$\frac{}{\Delta \mid x \vdash x \rightsquigarrow x}\ [\textsc{Var}]$$

# Declarative linear resource rules

borrowed owned input output

**Context** $\overbrace{\Delta}^{\text{borrowed}} \mid \overbrace{\Gamma}^{\text{owned}} \vdash \overbrace{e}^{\text{input}} \rightsquigarrow \overbrace{e'}^{\text{output}}$

own and only own it exactly once

consume a resource

$$\frac{}{\Delta \mid x \vdash x \rightsquigarrow x}\ [\textsc{Var}]$$

$$\frac{\varnothing \mid ys, x \vdash e \rightsquigarrow e' \qquad ys = \mathsf{fv}(\lambda x.\ e)}{\Delta \mid ys \vdash \lambda x.\ e \rightsquigarrow \lambda^{ys}\ x.\ e'}\ [\textsc{Lam}]$$

# Declarative linear resource rules

input output

borrowed owned

Context $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

own and only own it exactly once

consume a resource

$$\frac{}{\Delta \mid x \;\vdash\; x \;\rightsquigarrow\; x}\;[\textsc{Var}]$$

$$\frac{\varnothing \mid ys, x \;\vdash\; e \;\rightsquigarrow\; e' \qquad ys \;=\; \mathsf{fv}(\lambda x.\ e)}{\Delta \mid ys \;\vdash\; \lambda x.\ e \;\rightsquigarrow\; \lambda^{ys}\ x.\ e'}\;[\textsc{Lam}]$$

own all free variables

# Declarative linear resource rules

$$\text{Context}\quad \overbrace{\Delta}^{\text{borrowed}} \mid \overbrace{\Gamma}^{\text{owned}} \vdash \overbrace{e}^{\text{input}} \rightsquigarrow \overbrace{e'}^{\text{output}}$$

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \text{dup } x;\ e'}\ [\textsc{dup}]$$

$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \text{drop } x;\ e'}\ [\textsc{drop}]$$

# Declarative linear resource rules

Context $\overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash \overset{\text{input}}{e} \rightsquigarrow \overset{\text{output}}{e'}$

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \text{dup } x;\ e'}\ [\textsc{dup}]$$

explicit dup

$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \text{drop } x;\ e'}\ [\textsc{drop}]$$

# Declarative linear resource rules

$$\text{Context}\quad \overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash \overset{\text{input}}{e} \leadsto \overset{\text{output}}{e'}$$

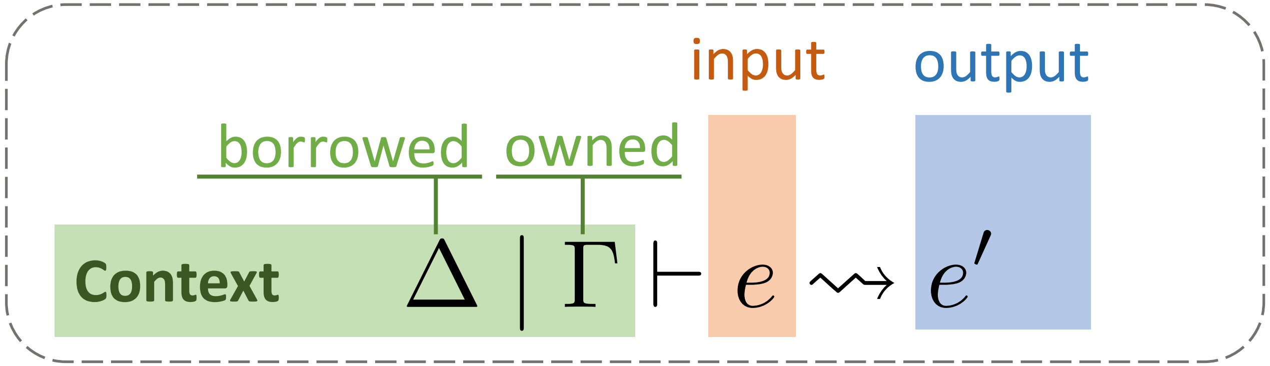
borrowed or owned

$$\frac{\Delta \mid \Gamma, x \;\vdash\; e \;\leadsto e' \qquad x \;\in \Delta, \Gamma}{\Delta \mid \Gamma \;\vdash\; e \;\leadsto \mathsf{dup}\; x;\; e'}\;[\textsc{dup}]$$

explicit dup

$$\frac{\Delta \mid \Gamma \;\vdash\; e \;\leadsto e'}{\Delta \mid \Gamma, x \;\vdash\; e \;\leadsto \mathsf{drop}\; x;\; e'}\;[\textsc{drop}]$$

# Declarative linear resource rules




$$\Delta \mid \Gamma \vdash e \rightsquigarrow e'$$

borrowed or owned

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \mathsf{dup}\ x;\ e'}\ [\textsc{dup}]$$

explicit dup

$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \mathsf{drop}\ x;\ e'}\ [\textsc{drop}]$$

# Declarative linear resource rules

input output

borrowed owned

Context $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

borrowed or owned

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \text{dup}\ x;\ e'}\ [\textsc{dup}]$$

explicit dup

$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \text{drop}\ x;\ e'}\ [\textsc{drop}]$$

explicit drop

# Declarative linear resource rules

borrowed owned input output

**Context** $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

borrowed or owned

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \mathsf{dup}\ x;\ e'}\ [\textsc{dup}]$$

explicit dup

$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \mathsf{drop}\ x;\ e'}\ [\textsc{drop}]$$

explicit drop

# Declarative linear resource rules

$$\text{Context}\quad \overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash \overset{\text{input}}{e} \rightsquigarrow \overset{\text{output}}{e'}$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{App}]$$

# Declarative linear resource rules

$$\underbrace{\overbrace{\Delta}^{\text{borrowed}} \mid \overbrace{\Gamma}^{\text{owned}}}_{\text{Context}} \vdash \overbrace{e}^{\text{input}} \rightsquigarrow \overbrace{e'}^{\text{output}}$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \underbrace{\Gamma_1}_{\text{split the owned context}}, \Gamma_2 \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{App}]$$

# Declarative linear resource rules

borrowed owned input output

Context $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$

$$\frac{\Delta, \Gamma_2 \mid \boxed{\Gamma_1} \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \boxed{\Gamma_2} \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \boxed{\Gamma_1}, \boxed{\Gamma_2} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{App}]$$

split the owned context

# Declarative linear resource rules

$$\underbrace{\overbrace{\Delta}^{\text{borrowed}} \mid \overbrace{\Gamma}^{\text{owned}}}_{\text{Context}} \vdash \overbrace{e}^{\text{input}} \rightsquigarrow \overbrace{e'}^{\text{output}}$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{App}]$$

split the owned context

# Perceus as syntax-directed linear resource rules

input output

borrowed owned

Context $\Delta \mid \Gamma \vdash_s e \rightsquigarrow e'$

# Perceus as syntax-directed linear resource rules

$$\text{Context}\quad \overset{\text{borrowed}}{\Delta} \mid \overset{\text{owned}}{\Gamma} \vdash_s \overset{\text{input}}{e} \rightsquigarrow \overset{\text{output}}{e'}$$

invariants

$\Delta \cap \Gamma \;=\; \varnothing$

multiplicity of each member in $\Delta, \Gamma$ is 1

$\Gamma \subseteq \mathsf{fv}(e) \qquad \mathsf{fv}(e) \subseteq \Delta, \Gamma$

# Perceus as syntax-directed linear resource rules

input output

borrowed owned

**Context** $\Delta \mid \Gamma \vdash_s e \rightsquigarrow e'$

invariants

$\Delta \cap \Gamma = \varnothing$

multiplicity of each member in $\Delta, \Gamma$ is 1

$\Gamma \subseteq \mathsf{fv}(e) \quad \mathsf{fv}(e) \subseteq \Delta, \Gamma$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{App}]$$

split the owned context

# Perceus as syntax-directed linear resource rules

borrowed owned

input output

Context $\Delta \mid \Gamma \vdash_s e \rightsquigarrow e'$

invariants

$$\Delta \cap \Gamma = \varnothing$$

multiplicity of each member in $\Delta, \Gamma$ is 1

$$\Gamma \subseteq \mathsf{fv}(e) \qquad \mathsf{fv}(e) \subseteq \Delta, \Gamma$$

split the owned context

$$\frac{\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_s e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \vdash_s e_2 \rightsquigarrow e_2' \qquad \Gamma_2 = \Gamma \cap \mathsf{fv}(e_2)}{\Delta \mid \Gamma \vdash_s e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{sapp}]$$

# Perceus as syntax-directed linear resource rules

borrowed owned input output

**Context** $\Delta \mid \Gamma \vdash_s e \rightsquigarrow e'$

invariants

$\Delta \cap \Gamma = \varnothing$

multiplicity of each member in $\Delta, \Gamma$ is 1

$\Gamma \subseteq \mathsf{fv}(e) \quad \mathsf{fv}(e) \subseteq \Delta, \Gamma$

not used

$$\frac{\boxed{x \notin \mathsf{fv}(e)} \quad \varnothing \mid ys \vdash_s e \rightsquigarrow e' \qquad ys = \mathsf{fv}(\lambda x.\, e) \quad \Delta_1 = ys - \Gamma}{\Delta, \Delta_1 \mid \Gamma \vdash_s \lambda x.\, e \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\, x.\, (\mathsf{drop}\ x;\ e')}\ [\textsc{slam-drop}]$$

# Reference-counted heap semantics

$$H \mid e \longrightarrow_r H' \mid e'$$

# Reference-counted heap semantics

input

$$\boxed{H \mid e} \longrightarrow_r H' \mid e'$$

# Reference-counted heap semantics

input

$$H \mid e \longrightarrow_r H' \mid e'$$

output

# Reference-counted heap semantics

input

$$H \mid e \longrightarrow_r H' \mid e'$$

output

Resources allocated in heap

| | | | | | |
|---|---|---|---|---|---|
| $(lam_r)$ | H $\mid (\lambda^{ys}\ x.\ e)$ | $\longrightarrow_r$ | H, $f \mapsto^1 \lambda^{ys} x.\ e$ | $\mid\ f$ | fresh $f$ |
| $(con_r)$ | H $\mid C\ x_1 \ldots x_n$ | $\longrightarrow_r$ | H, $z \mapsto^1\ C\ x_1 \ldots x_n$ | $\mid\ z$ | fresh $z$ |

# Reference-counted heap semantics

input

$$H \mid e \longrightarrow_r H' \mid e'$$

output

Resources allocated in heap

| | | | | | |
|---|---|---|---|---|---|
| $(lam_r)$ | H \| $(\lambda^{ys}\ x.\ e)$ | $\longrightarrow_r$ | H, $f \mapsto^1 \lambda^{ys} x.\ e$ | \| $f$ | fresh $f$ |
| $(con_r)$ | H \| $C\ x_1 \ldots x_n$ | $\longrightarrow_r$ | H, $z \mapsto^1\ C\ x_1 \ldots x_n$ | \| $z$ | fresh $z$ |

Beta rules

| | | | | |
|---|---|---|---|---|
| $(app_r)$ | H \| $f\ z$ | $\longrightarrow_r$ | H \| dup $ys$; drop $f$; $e[x{:=}z]$ | $(f \mapsto^n \lambda^{ys} x.\ e) \in$ H |
| $(match_r)$ | H \| match $x$ $\{\overline{p_i \to e_i}\}$ | $\longrightarrow_r$ | H \| dup $ys$; drop $x$; $e_i[xs{:=}ys]$ | with $p_i = C\ xs$ and $(x \mapsto^n C\ ys) \in$ H |
| $(bind_r)$ | H \| val $x = y$ ; $e$ | $\longrightarrow_r$ | H \| $e[x{:=}y]$ | |

# Reference-counted heap semantics

input

$$H \mid e \longrightarrow_r H' \mid e'$$

output

Resources allocated in heap

$$
\begin{array}{llcll}
(lam_r) & \mathsf{H} \mid (\lambda^{ys}\ x.\ e) & \longrightarrow_r & \mathsf{H}, f \mapsto^1 \lambda^{ys} x.\ e \quad \mid f & \text{fresh } f \\
(con_r) & \mathsf{H} \mid C\ x_1 \ldots x_n & \longrightarrow_r & \mathsf{H}, z \mapsto^1 C\ x_1 \ldots x_n \quad \mid z & \text{fresh } z
\end{array}
$$

Beta rules

$$
\begin{array}{llcll}
(app_r) & \mathsf{H} \mid f\ z & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ f;\ e[x{:=}z] & (f \mapsto^n \lambda^{ys} x.\ e) \in \mathsf{H} \\
(match_r) & \mathsf{H} \mid \mathsf{match}\ x\ \{\overline{p_i \to e_i}\} & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ x;\ e_i[xs{:=}ys] & \text{with } p_i = C\ xs \text{ and } (x \mapsto^n C\ ys) \in \mathsf{H} \\
(bind_r) & \mathsf{H} \mid \mathsf{val}\ x = y\ ;\ e & \longrightarrow_r & \mathsf{H} \mid e[x{:=}y] &
\end{array}
$$

Reference counting instructions

$$
\begin{array}{lllcll}
(dup_r) & \mathsf{H}, x \mapsto^n v & \mid \mathsf{dup}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n+1} v \mid e & \\
(drop_r) & \mathsf{H}, x \mapsto^{n+1} v & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n} v \mid e & \text{if } n \geqslant 1 \\
(dlam_r) & \mathsf{H}, x \mapsto^1 \lambda^{ys} z.e & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ ys;\ e & \\
(dcon_r) & \mathsf{H}, x \mapsto^1 C\ ys & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ ys;\ e &
\end{array}
$$

# Perceus is precise and garbage free

# Perceus is precise and garbage free

reach $(x, H \mid e)$

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$

# Perceus is precise and garbage free

reach $(x, H \mid e)$

- $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^n v \in$ H $\quad$ reach$(x, H \mid v)$

# Perceus is precise and garbage free

reach $(x, H \mid e)$
- $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^n v \in$ H $\quad$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$

- reach $(y, H \mid e)$ $y \mapsto^n v \in \mathsf{H}$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_s e \rightsquigarrow e'$

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$

- reach $(y, H \mid e)$ $y \mapsto^n v \in \mathsf{H}$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$

$\varnothing \mid e' \longmapsto^*{}_r \mathsf{H} \mid x$

# Perceus is precise and garbage free

reach $(x, H \mid e)$
- $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^{n} v \in \mathrm{H} \quad$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$ Then

$\varnothing \mid e' \longmapsto^{*}{}_{r} \mathrm{H} \mid x$

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$

- reach $(y, H \mid e)$ $y \mapsto^n v \in \mathsf{H}$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$ Then for every intermediate state $\mathsf{H}_i \mid e_i$,

$\varnothing \mid e' \longmapsto^*{}_r \mathsf{H} \mid x$

# Perceus is precise and garbage free

reach $(x, H \mid e)$

- $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^n v \in \mathsf{H}$ $\quad$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$ $\quad$ Then $\quad$ for every intermediate state $\mathsf{H}_i \mid e_i$,

$\varnothing \mid e' \longmapsto^{*}{}_{r} \mathsf{H} \mid x$ $\quad$ which is not at a rc operation,

# Perceus is precise and garbage free

reach $(x, H \mid e)$
- $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^n v \in \mathsf{H}$ $\quad$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$
$\varnothing \mid e' \longmapsto^{*}{}_{r} \mathsf{H} \mid x$

Then for every intermediate state $\mathsf{H}_i \mid e_i$,
which is not at a rc operation,
for all $y \in \mathsf{dom}(\mathsf{H}_i)$

# Perceus is precise and garbage free

reach $(x, H \mid e)$

- $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^n v \in \mathsf{H} \quad$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$
$\varnothing \mid e' \longmapsto^*{}_r \mathsf{H} \mid x$

Then for every intermediate state $\mathsf{H}_i \mid e_i$,
which is not at a rc operation,
for all $y \in \mathrm{dom}(\mathsf{H}_i)$
reach $(y, \mathsf{H}_i \mid \lceil e_i \rceil)$

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $y \mapsto^n v \in \mathsf{H}$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_s e \rightsquigarrow e'$
$\varnothing \mid e' \longmapsto^*_r \mathsf{H} \mid x$

Then for every intermediate state $\mathsf{H}_i \mid e_i$,
which is not at a rc operation,
for all $y \in \mathsf{dom}(\mathsf{H}_i)$
reach $(y,\ \mathsf{H}_i \mid \lceil e_i \rceil)$

erase rc operations

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $\quad y \mapsto^n v \in \mathsf{H} \quad$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_s e \rightsquigarrow e'$
$\varnothing \mid e' \longmapsto^*{}_r \mathsf{H} \mid x$

Then for every intermediate state $\mathsf{H}_i \mid e_i$,
which is not at a rc operation,
for all $y \in \mathrm{dom}(\mathsf{H}_i)$
reach $(y, \mathsf{H}_i \mid \lceil e_i \rceil)$

erase rc operations

$$y \mapsto^1 () \mid (\lambda x.\, x)\ (\mathsf{drop}\ y;\ ())$$ ✗

# Perceus is precise and garbage free

reach $(x, H \mid e)$ - $x \in \mathsf{fv}(e)$
- reach $(y, H \mid e)$ $y \mapsto^n v \in \mathsf{H}$ reach$(x, H \mid v)$

**Theorem 4.** (*Perceus is precise and garbage free*)

Given $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$
$\varnothing \mid e' \longmapsto^{*}{}_{r} \mathsf{H} \mid x$

Then for every intermediate state $\mathsf{H}_i \mid e_i$,
which is not at a rc operation,
for all $y \in \mathsf{dom}(\mathsf{H}_i)$
reach $(y,\ \mathsf{H}_i \mid \lceil e_i \rceil)$

erase rc operations

$y \mapsto^1 () \mid (\lambda x.\ x)\ (\mathsf{drop}\ y;\ ())$ ✗

$y \mapsto^1 () \mid \mathsf{drop}\ y;\ (\lambda x.\ x)\ ()$ ✓

# Summary

② Koka 101

③ Functional But In-Place (FBIP)

① Perceus



④ Linear Resource Calculus $\lambda^1$

# Summary

② Koka 101

③ Functional But In-Place (FBIP)

① Perceus

④ Linear Resource Calculus $\lambda^1$

# Summary

② Koka 101

③ Functional But In-Place (FBIP)

① Perceus

④ Linear Resource Calculus $\lambda^1$

# Summary

② Koka 101

③ Functional But In-Place (FBIP)

① Perceus

④ Linear Resource Calculus $\lambda^1$

# Summary

② Koka 101

③ Functional But In-Place (FBIP)

① Perceus

④ Linear Resource Calculus $\lambda^1$

# Benchmarks

Koka | Koka,no-opt | OCaml | Haskell
Swift | Java | C++

# Benchmarks

Koka | Koka,no-opt | OCaml | Haskell
Swift | Java | C++

**Goal**: Perceus is viable and can be competitive.

# Benchmarks

Koka Koka,no-opt OCaml Haskell
Swift Java C++

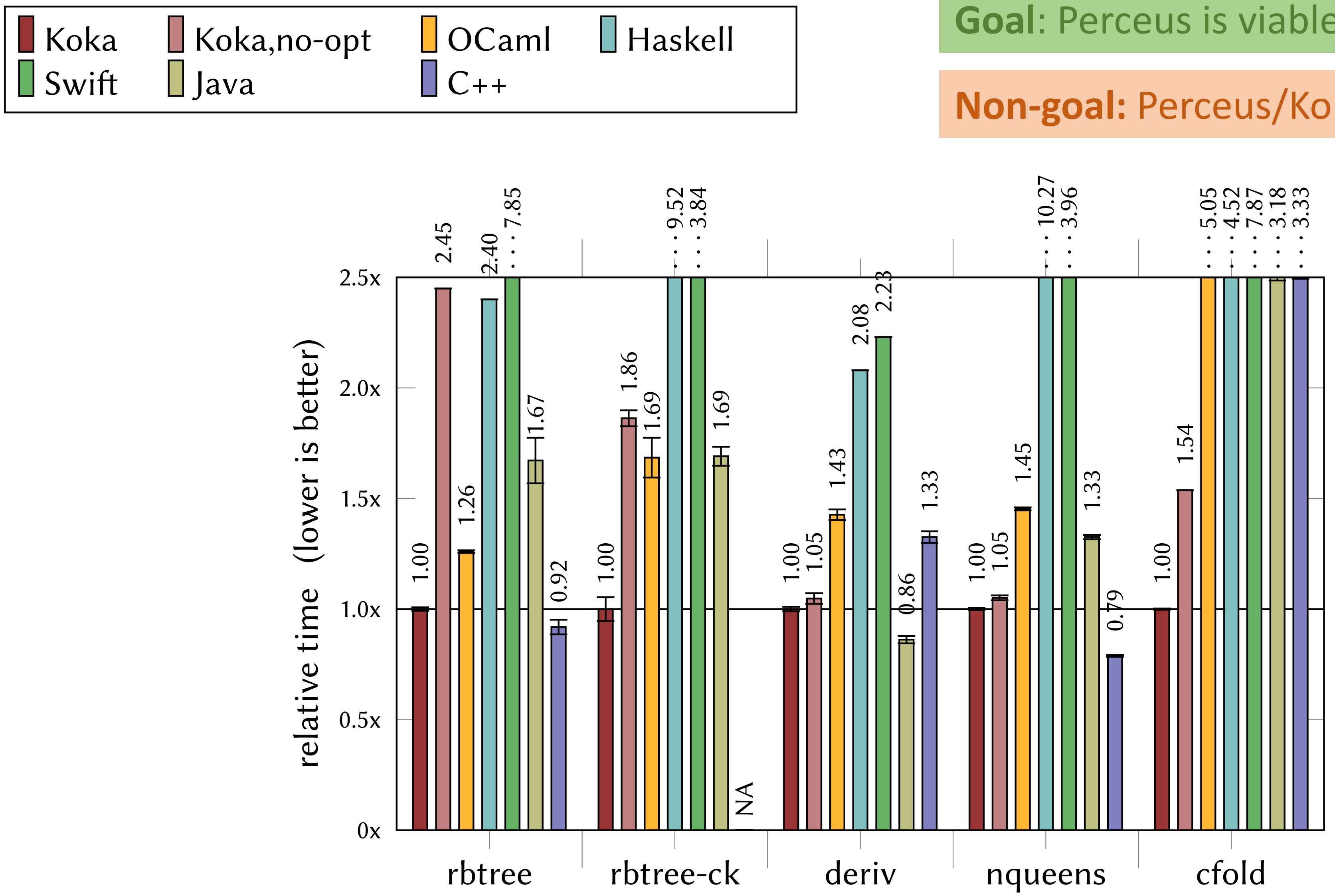
**Goal**: Perceus is viable and can be competitive.

**Non-goal:** Perceus/Koka is the best!

# Benchmarks

**Goal**: Perceus is viable and can be competitive.

**Non-goal:** Perceus/Koka is the best!



# Perceus

## Garbage Free Reference Counting with Reuse


Ningning Xie

香 港 大 學

THE UNIVERSITY OF HONG KONG

Joint work with Alex Reinking,
Leonardo de Moura, and Daan Leijen

Microsoft Research