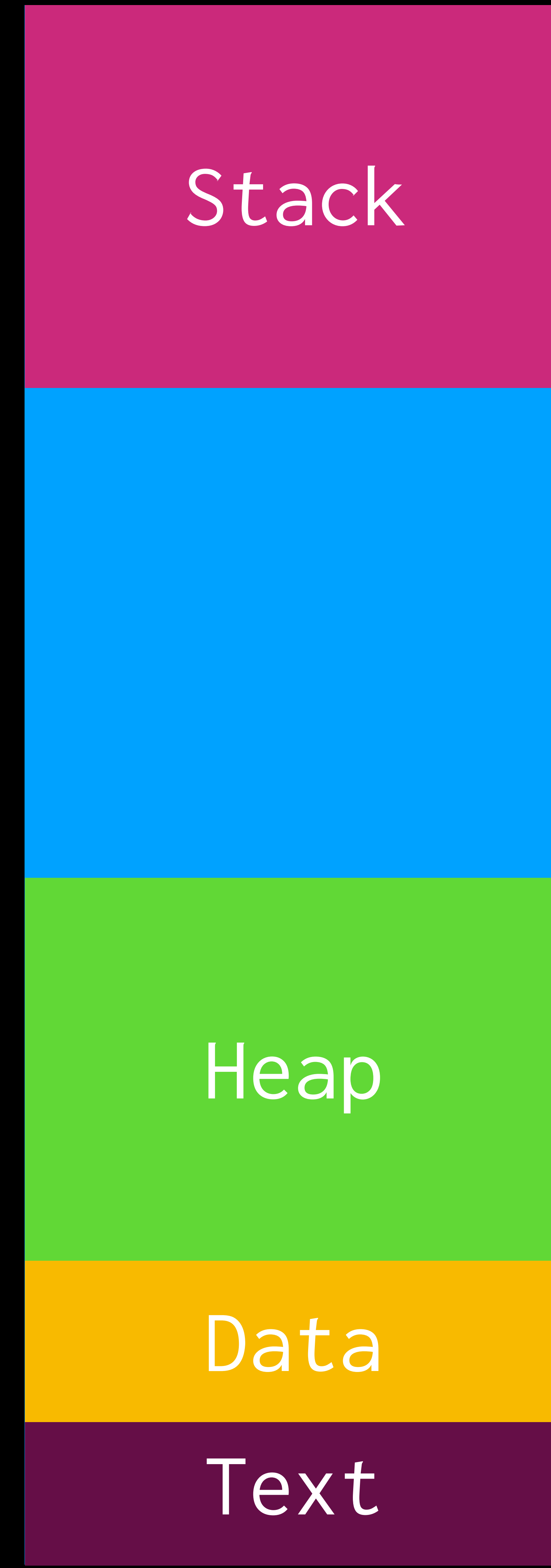
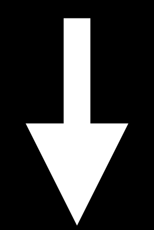
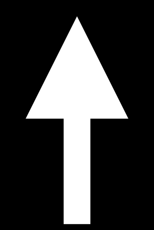
memory


Stack
Heap
Data
Text



Q: how to manage memory?

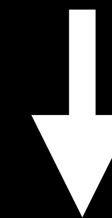
lexical scoping

```
struct foo {
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;
int z;
{
    struct foo sf;
    f(&sf);
    {
        int w = 1;
        sf.f2 = w;
        z += sf.f1;
    }
}
return z;
```

lexical scoping

```
struct foo {                      usage (B)
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;                        +4
int z;                            +4
{
    struct foo sf;                +8
    f(&sf);
    {
        int w = 1;                +4
        sf.f2 = w;
        z += sf.f1;
    }                             -4
}                                 -8
return z;                         -8
```

Stack



lexical scoping

```
struct foo {
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;
int z;
{
    struct foo sf;
    f(&sf);
    {
        int w = 1;
        sf.f2 = w;
        z += sf.f1;
    }
}
return z;
```

“stack discipline”

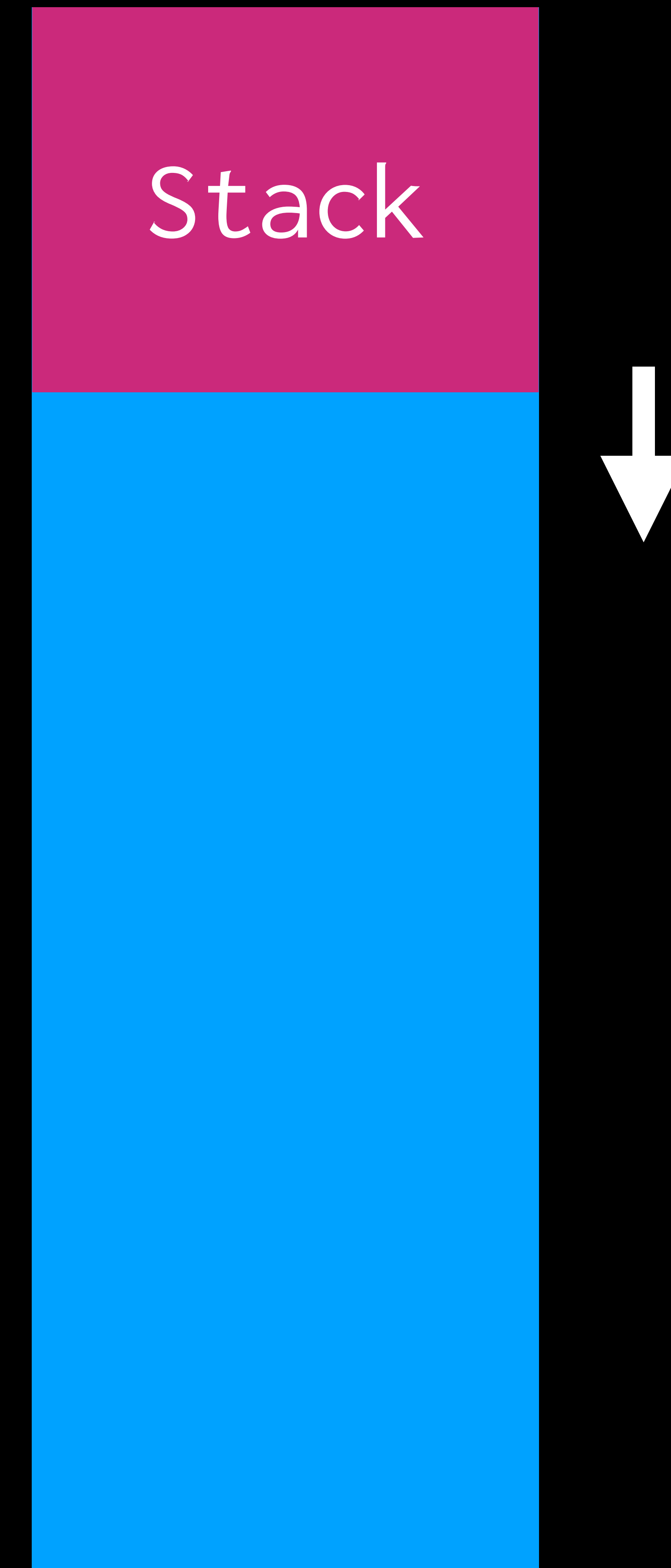
allocation is as easy as decrementing stack pointer!

manual

```c
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return alloca(sizeof(struct foo));
}

struct foo *sf = f();
return sf->x1;
```

manual

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return alloca(sizeof(struct foo));
}

struct foo *sf = f();
return sf->x1;
```



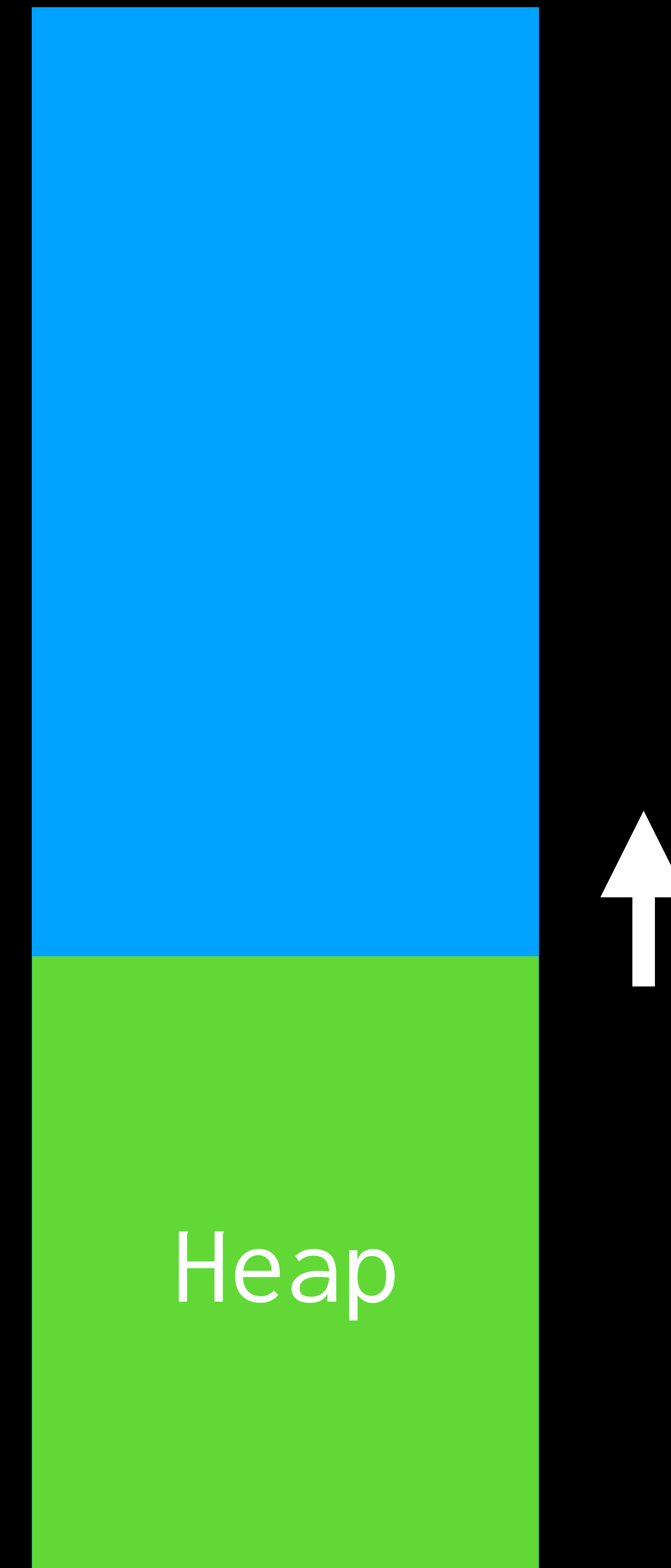
manual

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return alloca(sizeof(struct foo));
}

struct foo *sf = f();
return sf->x1;
```

clobbered!

manual

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return malloc(sizeof(struct foo));
}

struct foo *sf = f();
return sf->x1;
```



manual

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return malloc(sizeof(struct foo));
}

struct foo *sf = f();
return sf->x1;
```

how is this implemented?

manual

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return malloc(sizeof(struct foo));
}

struct foo *sf = f();
return sf->x1;
```

how much heap are we still using?

manual

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return malloc(sizeof(struct foo));
}

struct foo *sf = f();
int retval = sf->x1;
free(sf);
return retval;
```

manual

```c
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return malloc(sizeof(struct foo));
}

struct foo *sf = f();
int retval = sf->x1;
free(sf);
return retval;
```

what a hassle!

reference counting

```
struct foo {
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;
int z;
{
    struct foo *sf = rc_alloc(sizeof(struct foo));
    f(sf);
    {
        int w = 1;
        sf.f2 = w;
        z += sf.f1;
    }

}
return z;
```

reference counting

```
struct foo {
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;
int z;
{
    struct foo *sf = rc_alloc(sizeof(struct foo));
    f(sf);
    {
        int w = 1;
        sf.f2 = w;
        z += sf.f1;
    }
    rc_free(sf);
}
return z;
```

inserted by the compiler!

reference counting

```
struct foo {
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;
int z;
{
    struct foo *sf = rc_alloc(sizeof(struct foo));
    f(sf);
    {
        int w = 1;
        sf.f2 = w;
        z += sf.f1;
    }
    rc_free(sf);
}
return z;
```

reference counting

```
struct foo {
    int x1;
    int x2;
};

void f(struct foo *sf);

int x = 1;
int z;
{
    struct foo *sf = rc_alloc(sizeof(struct foo));
    f(sf);
    {
        int w = 1;
        sf.f2 = w;
        z += sf.f1;
    }
    rc_free(sf);
}
return z;
```

exactly same principle
as stack allocation

reference counting

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return rc_alloc(sizeof(struct foo));
}

struct foo *sf = f();
int retval = sf->x1;

return retval;
```

reference counting

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return rc_alloc(sizeof(struct foo));
}

struct foo *sf = f();
int retval = sf->x1;

return retval;
```

uh-oh…
deallocate?

reference counting

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return rc_retain(rc_alloc(sizeof(struct foo)));
}

struct foo *sf = f();
int retval = sf->x1;
rc_release(sf);
return retval;
```

reference counting

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return rc_retain(rc_alloc(sizeof(struct foo)));
}

struct foo *sf = f();
int retval = sf->x1;
rc_release(sf);
return retval;
```

retain: increment refcount

release: decrement refcount
if zero, free

reference counting

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {       rc: 0
    return rc_retain(rc_alloc(sizeof(struct foo)));
}                   rc: 1
          rc: 1
struct foo *sf = f();
int retval = sf->x1;
rc_release(sf);  rc: 0      freed!
return retval;
```

reference counting

```
struct foo {
    int x1;
    int x2;
};

struct foo *f() {
    return rc_retain(rc_alloc(sizeof(struct foo)));
}

struct foo *sf = f();
int retval = sf->x1;
rc_release(sf);
return retval;
```

in practice: compiler generates retain/release

reference cycles

```
struct foo {
    struct foo *f1;
};

void chain(struct foo *sf1, struct foo *sf2) {
    sf1->f1 = rc_retain(sf2);
    sf2->f1 = rc_retain(sf1);
}

void f() {
    struct foo *sf1 = rc_retain(rc_alloc(sizeof(struct foo)));
    struct foo *sf2 = rc_retain(rc_alloc(sizeof(struct foo)));
    chain(sf1, sf2);
    rc_release(sf1);
    rc_release(sf2);
}

f();
```

reference cycles

```
struct foo {
    struct foo *f1;
};

void chain(struct foo *sf1, struct foo *sf2) {
    sf1->f1 = rc_retain(sf2);
    sf2->f1 = rc_retain(sf1);
}

void f() {
    struct foo *sf1 = rc_retain(rc_alloc(sizeof(struct foo)));    rc: 1
    struct foo *sf2 = rc_retain(rc_alloc(sizeof(struct foo)));    rc: 1
    chain(sf1, sf2);
    rc_release(sf1);
    rc_release(sf2);
}

f();
```

reference cycles

```
struct foo {
    struct foo *f1;
};

void chain(struct foo *sf1, struct foo *sf2) {
    sf1->f1 = rc_retain(sf2);    rc: 2
    sf2->f1 = rc_retain(sf1);    rc: 2
}

void f() {
    struct foo *sf1 = rc_retain(rc_alloc(sizeof(struct foo)));    rc: 1
    struct foo *sf2 = rc_retain(rc_alloc(sizeof(struct foo)));    rc: 1
    chain(sf1, sf2);
    rc_release(sf1);
    rc_release(sf2);
}

f();
```

reference cycles

```
struct foo {
    struct foo *f1;
};

void chain(struct foo *sf1, struct foo *sf2) {
    sf1->f1 = rc_retain(sf2);     rc: 2
    sf2->f1 = rc_retain(sf1);     rc: 2
}

void f() {
    struct foo *sf1 = rc_retain(rc_alloc(sizeof(struct foo)));    rc: 1
    struct foo *sf2 = rc_retain(rc_alloc(sizeof(struct foo)));    rc: 1
    chain(sf1, sf2);
    rc_release(sf1);    rc: 1
    rc_release(sf2);    rc: 1
}                      leaked!

f();
```

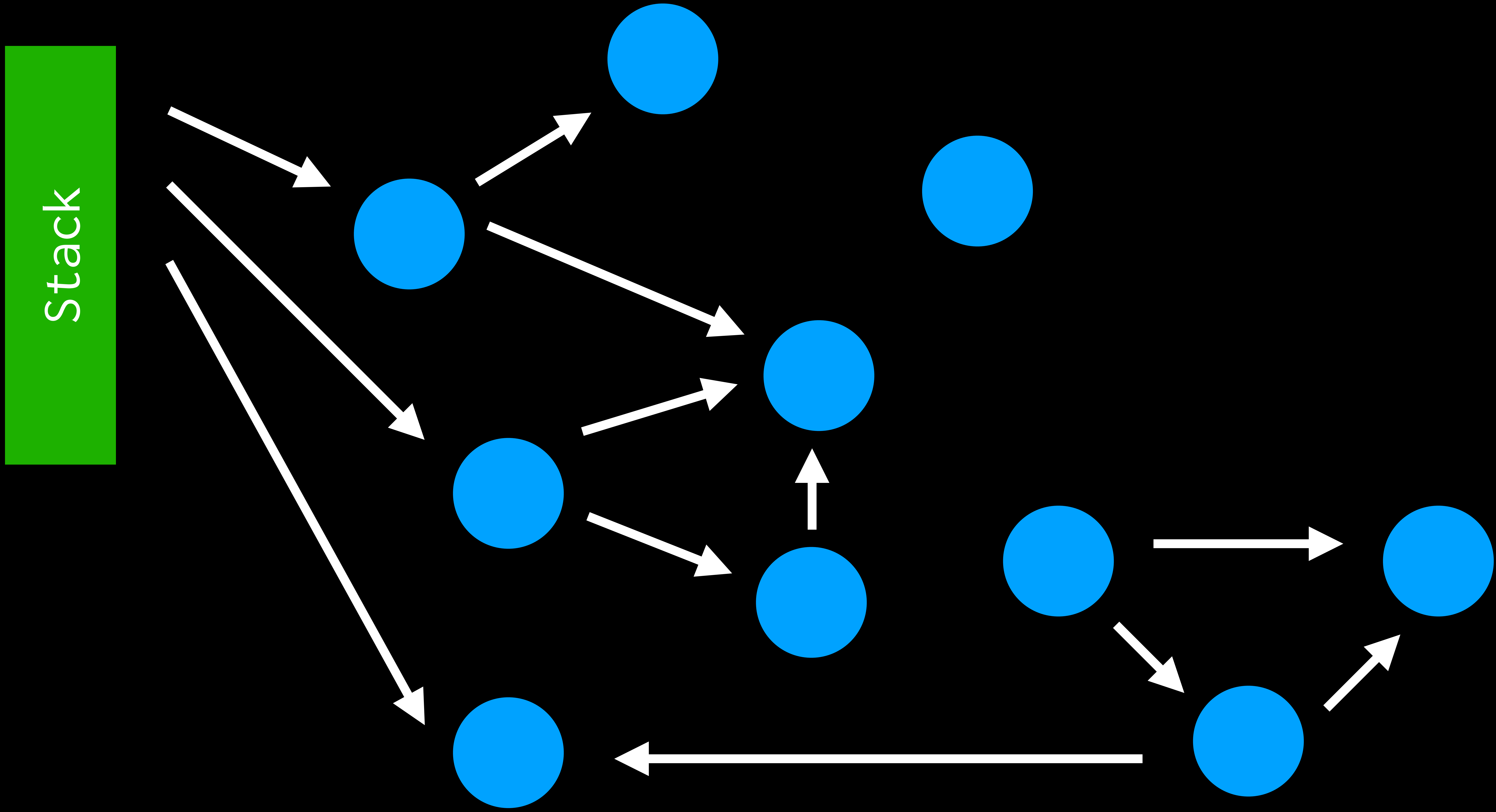
reference cycles

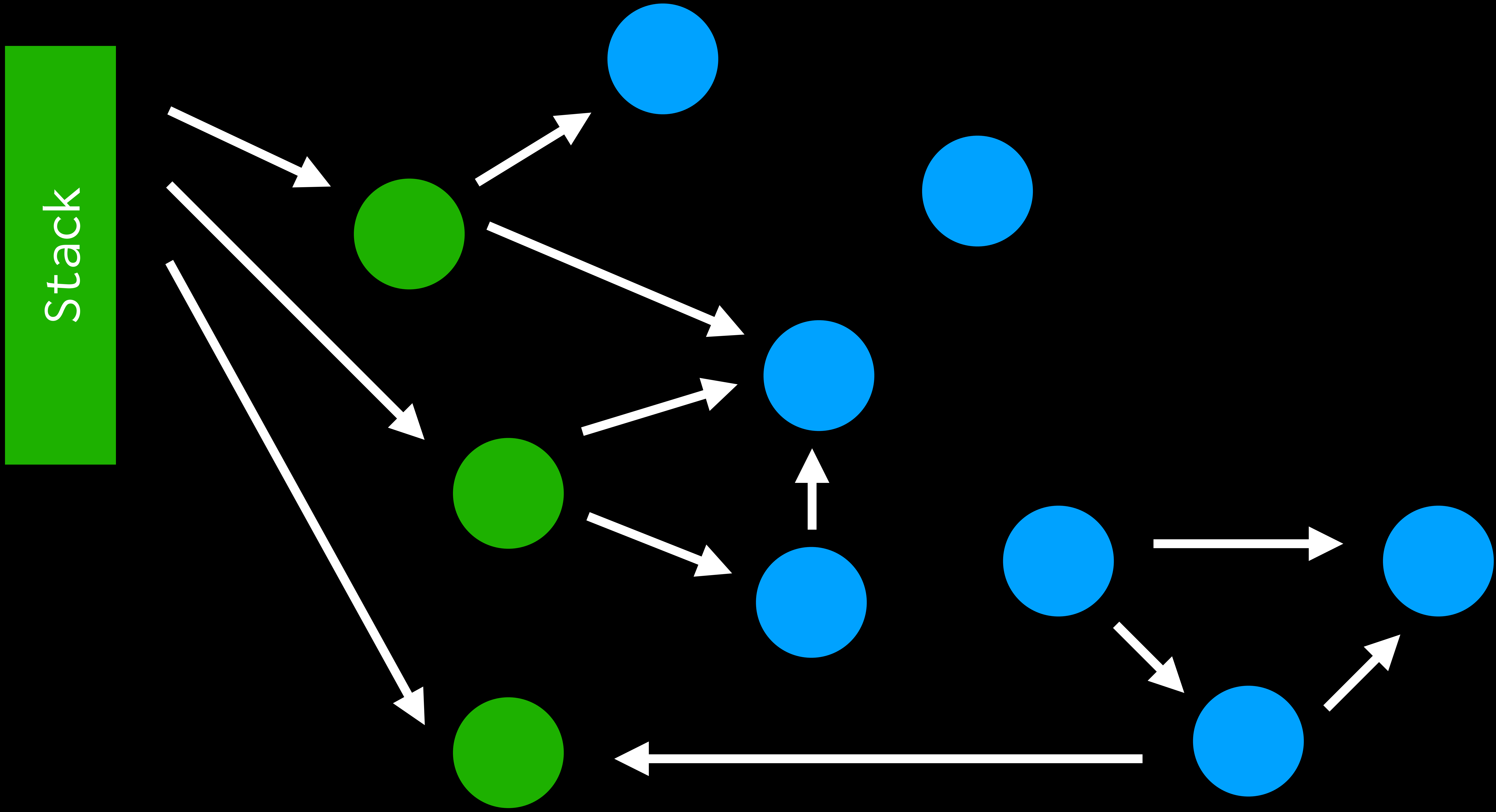
```
struct foo {
    struct foo *f1;
};

void chain(struct foo *sf1, struct foo *sf2) {
    sf1->f1 = rc_retain(sf2);
    sf2->f1 = rc_retain(sf1);
}

void f() {
    struct foo *sf1 = rc_retain(rc_alloc(sizeof(struct foo)));
    struct foo *sf2 = rc_retain(rc_alloc(sizeof(struct foo)));
    chain(sf1, sf2);
    rc_release(sf1);
    rc_release(sf2);
}

f();
```

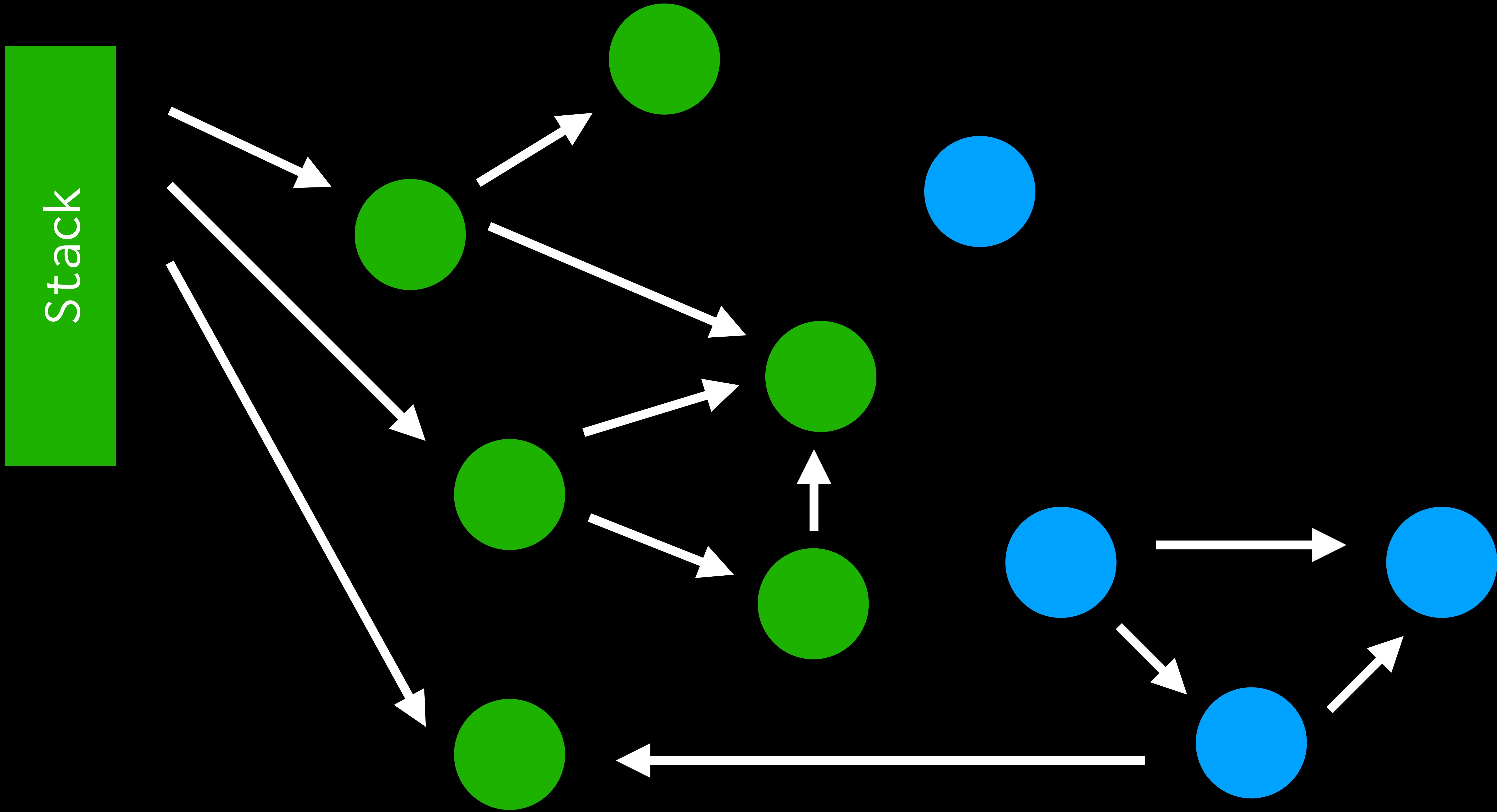
in practice: weak pointers

garbage collection

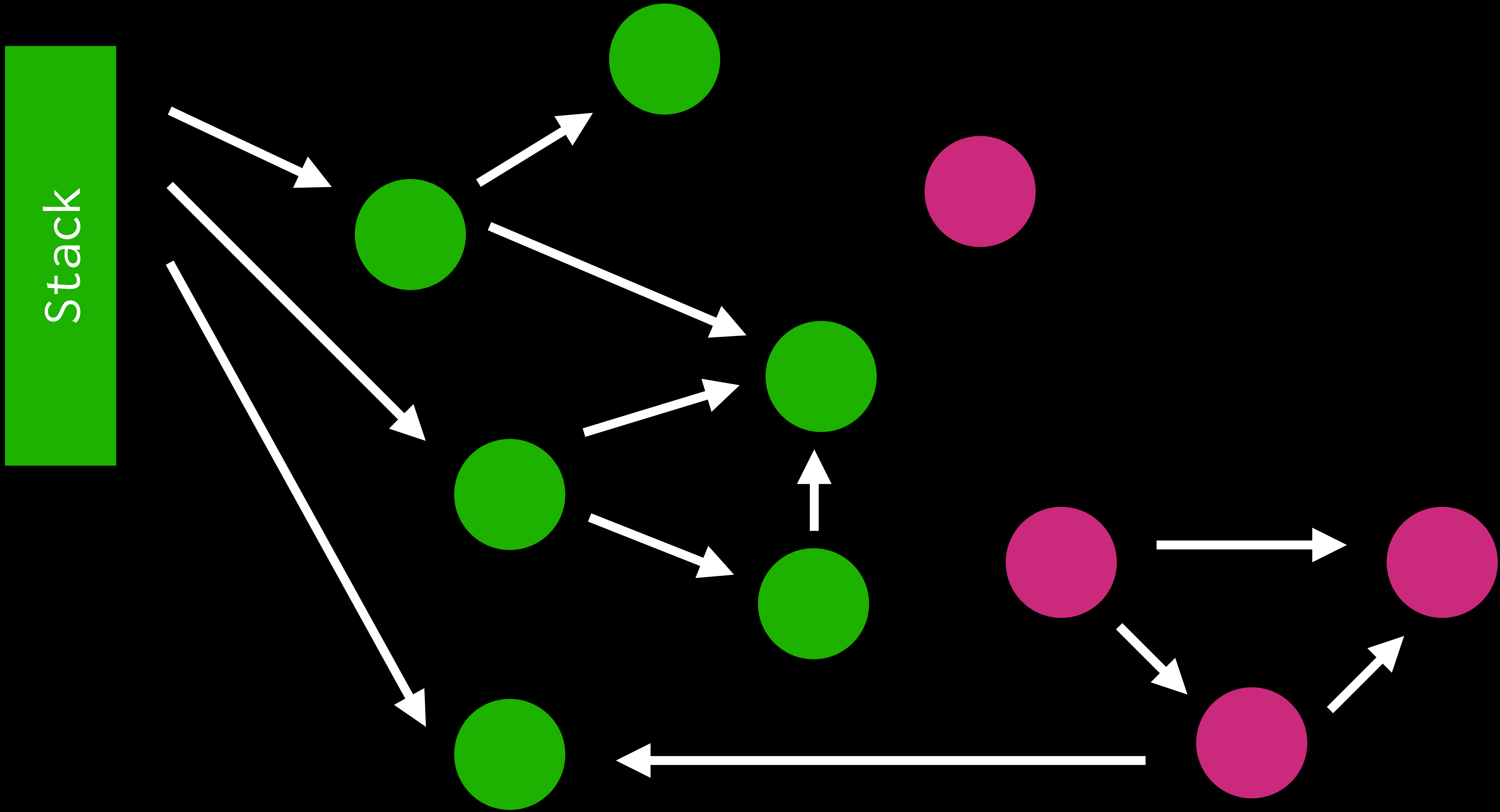
- traverse heap, looking for accessible cells
- start at roots accessible via stack
- mark and sweep: eliminate unaccessible cells
- alternatively, copy reachable heap, discard old
- not (directly) based on static analysis

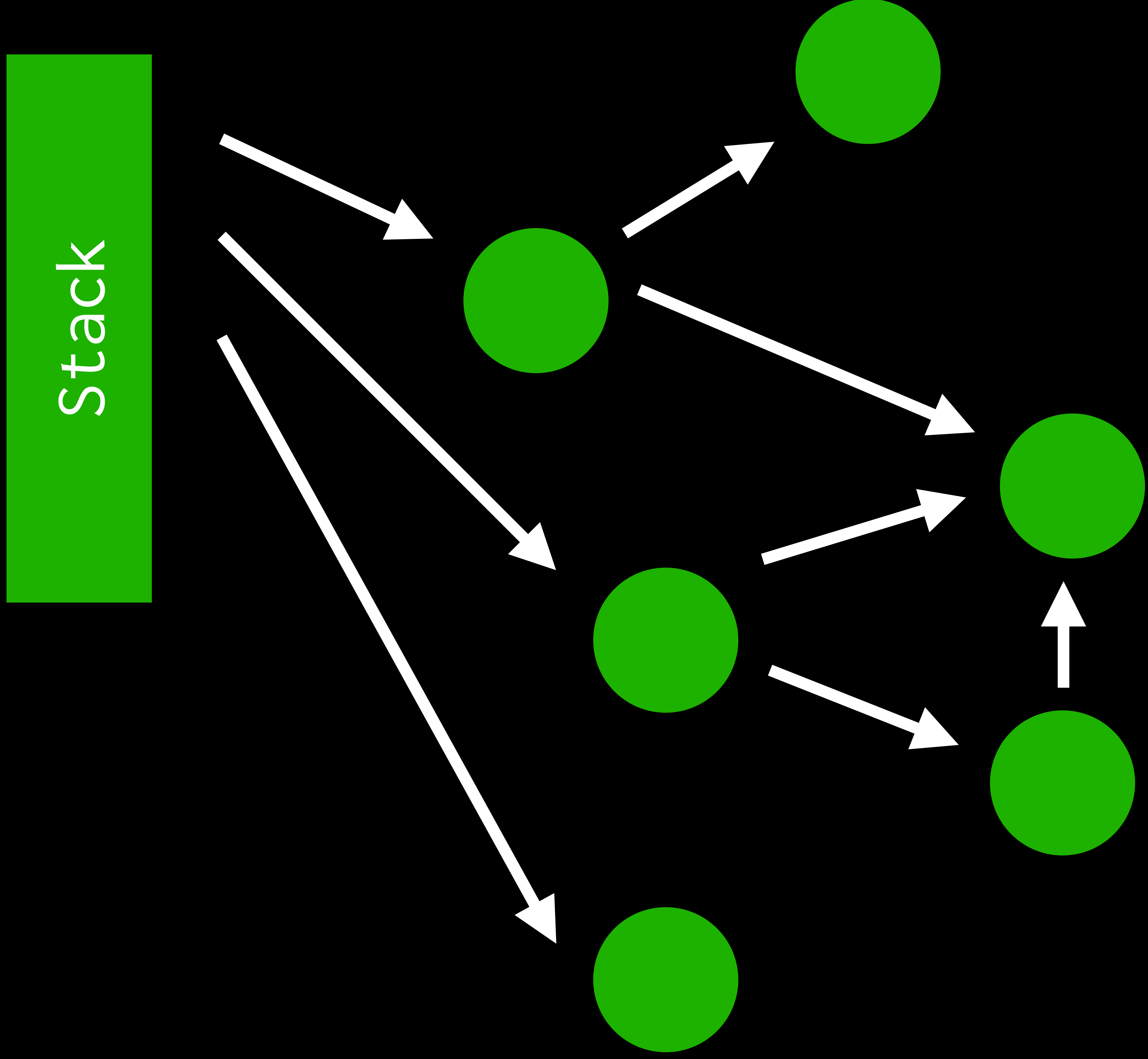

Stack


Stack



Stack


Stack


Stack

which is better?

garbage collection

- ML, Java, JavaScript, etc.
- free from the reference cycle problem
- potentially compacts the heap
- other advantages/problems?
- in practice, generations and hybrid schemes

despite its flaws, the world is GC.

...but can we do better?

Q: what made ref counting nontrivial?

A: sharing and cycles.

Q: why is sharing hard?

A: don’t statically know when to free.

ownership

key idea: *statically knowing when to free enables easy, deterministic memory management*

ownership

key idea: *statically knowing when to free enables easy, deterministic memory management*

GC and RC: conservative underestimate of data *ownership* at runtime

ownership

key idea: *statically knowing when to free enables easy, deterministic memory management*

GC and RC: conservative underestimate of data *ownership* at runtime

delayed free is practically never!

ownership

key idea: *statically knowing when to free enables easy, deterministic memory management*

GC and RC: conservative underestimate of data *ownership* at runtime

delayed free is practically never!

stack allocation isn’t general...so generalize it!

move semantics

each object has a *fixed owner*

sharing of a *unique type* is *statically disallowed*

values can be constructed and destructed, but not implicitly copied or structurally shared

see: std::unique_ptr<T> in C++, Box<T> in Rust

can be modeled formally in the λ-calculus

weakening

$$\frac{\Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_1, x_1 : \tau_1, \Gamma_2 \vdash e : \tau}$$

“useless premises are okay”

contraction

$$\frac{\Gamma_1, x_2 : \tau_1, x_3 : \tau_1, \Gamma_2 \vdash e : \tau_2}{\Gamma_1, x_1 : \tau_1, \Gamma_2 \vdash [x_2/x_1][x_3/x_1]e : \tau_2}$$

“what can be done with many, can be done with one”

linear types

substructural logic lacking *weakening* and *contraction*

like SML with unused variable checking, no sharing

in practice: *affine types* allow weakening, not contraction

```
val x = []
val y = 1::x
val z = x        (* this isn't allowed! *)
```

```
fun clone (xs : 'a list) : ('a list * 'a list) =
  (xs, xs)
```

```
fun clone (xs : 'a list) : ('a list * 'a list) =
  (xs, xs)
```

sharing isn't allowed

```
fun clone (xs : 'a list) (f : 'a -> 'a * 'a) =
  case xs of
    [] => ([], [])
  | x::xs =>
    let
      val (x1, x2) = f x
      val (xs1, xs2) = clone xs f
    in
      (x1::xs1, x2::xs2)
    end
```

why should cloning be explicit?

why should cloning be explicit?

accurate resource usage

why should cloning be explicit?

accurate resource usage

what does cloning a *file* do?

or a *network socket*?

zero allocation

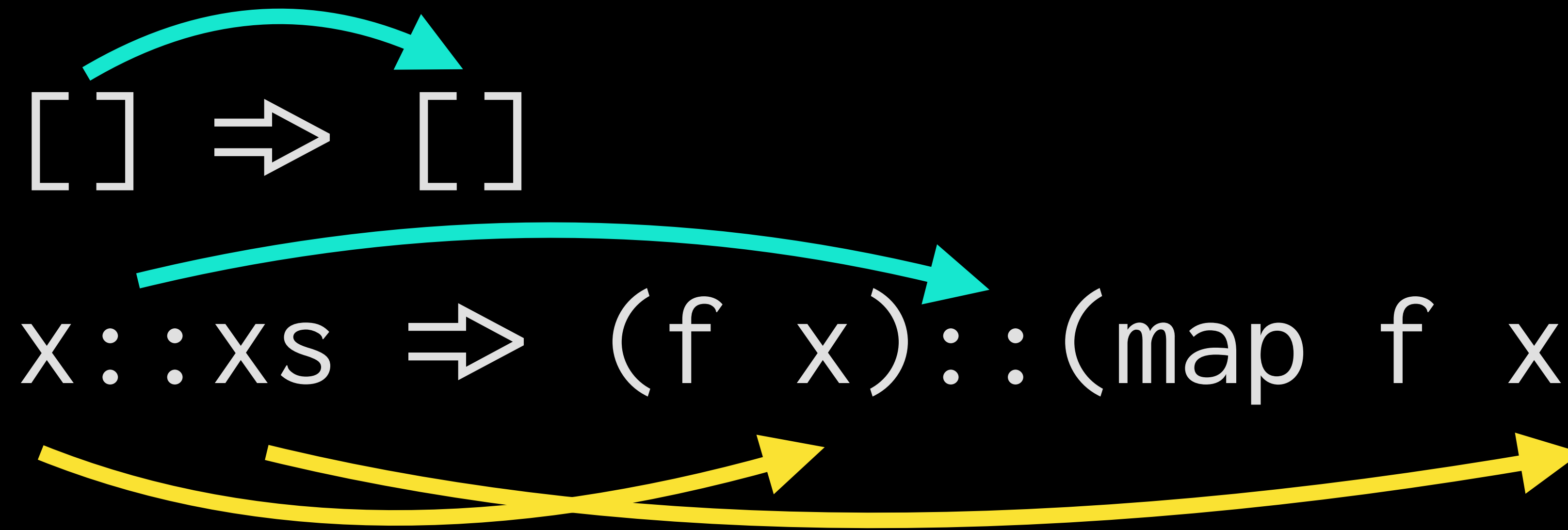
```
fun map (f : 'a -> 'b) (xs : 'a list) : 'b list =
  case xs of
    [] => []
  | x::xs => (f x)::(map f xs)
```

how many allocations?

zero allocation

linear type!

```
fun map (f : 'a -> 'b) (xs : 'a list) : 'b list =
  case xs of
    [] => []
  | x::xs => (f x)::(map f xs)
```



zero allocation

```
fun map (f : 'a -> 'b) (xs : 'a list) : 'b list =
  case xs of
    [] => []
  | x::xs => (f x)::(map f xs)
```

no allocations!

optimize to an *in-place for-loop*!

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = List.fold (fn ((i, x), acc) =>
      let
        val existing = case Table.get acc i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.set acc i new
      end
    ) Table.empty xs
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = List.fold (fn ((i, x), acc) =>
      let
        val existing = case Table.get acc i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.set acc i new
      end
    ) Table.empty xs
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

no sharing!

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = List.fold (fn ((i, x), acc) =>
      let
        val (existing, acc') = case Table.remove acc i of
            (NONE, acc') => ([], acc')
          | (SOME l, acc') => (l, acc')
        val new = x::existing
      in
        Table.add acc' i new
      end
    ) Table.empty xs
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = List.fold (fn ((i, x), acc) =>
      let
        val (existing, acc') = case Table.remove acc i of
            (NONE, acc') => ([], acc')
          | (SOME l, acc') => (l, acc')
        val new = x::existing
      in
        Table.add acc' i new
      end
    ) Table.empty xs
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

wow, that’s annoying

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.delete table i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert table i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

Q: isn’t imperative programming bad?

A: well, observable side effects, maybe.

```
val x = ref 1
val y = x
val z = y := 2
val 1 = !x          (* exception Bind *)
```

ML world

```
val x = ref 1
val y = x
val z = y := 2
val 1 = !x        Error: ‘x’ used after move
```

linear world

```
let l = [1, 2, 3];

l.push(4);
```

```
let l = [1, 2, 3];

  let l = l @ [4];
```

imperative = functional?

“Linear types can change the world!”

Phil Wadler, 1990

http://www.cs.ioc.ee/ewscs/2010/mycroft/linear-2up.pdf

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove table i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert table i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove table i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert table i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove table i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert table i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

really just state monad?

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove table i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert table i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

really just a *binding*

references

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \&e : \&\tau}$$

references are exempt from linearity

construction: tied to owned object

destruction: freely before owned object

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove (&table) i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert (&table) i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove (&table) i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert (&table) i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

predictable resource usage
\+ imperative efficiency
\+ almost no observable side effects

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove (&table) i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert (&table) i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove (&table) i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert (&table) i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

what if someone held (&table) here?

possible race condition!

data race freedom

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \&\texttt{mut}\ e : \&\texttt{mut}\ \tau}$$

mutable references are **not** (fully) **exempt** from linearity

construction: *only one* per object,
and not when any *immutable* refs exist!

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val mut table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove (&mut table) i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert (&mut table) i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

+ no observable side effects

```
fun aggregate (xs : (int * 'a) list) : (int * 'a list) list =
  let
    val mut table = Table.empty ()
    val _ = List.fold (fn ((i, x), _) =>
      let
        val existing = case Table.remove (&mut table) i of
            NONE => []
          | SOME l => l
        val new = x::existing
      in
        Table.insert (&mut table) i new
      end
    ) xs ()
  in
    Table.fold (fn ((i, xs), acc) => (i, xs)::acc) [] table
  end
```

Rust, in a nutshell!

regions

- how long can references last?

regions

- how long can references last?
- Swift: “as long as the ref count is above zero”

regions

- how long can references last?
- Swift: “as long as the ref count is above zero”
- ML: “as long as the object is reachable”

regions

- how long can references last?
- Swift: “as long as the ref count is above zero”
- ML: “as long as the object is reachable”
- C: “who knows?”

regions

- how long can references last?
- Swift: “as long as the ref count is above zero”
- ML: “as long as the object is reachable”
- C: “who knows?”
- Rust:
  a reference &’a T lasts as long as the *lifetime* ‘a

regions

- Ownership requires explicit allocation
- References must not outlast values
- Can ownership be inferred?
- Generalizing stack allocation fully: region types

regions

- Annotate values with region variables
- Lambdas form region closures
- Evaluation destroys regions
- Type safety: no dangling pointer is dereferenced
- Region annotation: meaning-preserving refinement
- At runtime: stack allocation, bump allocators, deterministic whole region deallocation
- Effect types

```
{ region <‘r1> r1;
  list_t<int,’r1> x = NULL;
  { region <‘r2> r2;
    list_t<int,’r2> y = rcopy(r2, x);
    x = rnew(r1) List(4,x);
    y = rnew(r2) List(5,y);
    // x = rnew(r2) List(4,x); // Error : incompatible regions
    // x = rnew(r1) List(4,y); // Error : List in unique region
  }
}
```

Cyclone

```
list_t<int,’r> fib_rec(region_t<‘r> r, int a, int b, int n)
{
  if (n == 0)
    return rnew(r) List(b, NULL);
  else {
    region <‘s> s;
    list_t<int, ‘s> z = fib_rec(s, b, a+b, n-1);
    z = rnew(s) List(b, z);
    return rcopy(r, z);
  }
}
list_t<int,’r> fib(region_t<‘r> r, int n) {
  return fib_rec(r,1,1,n-1);
}
```

Cyclone

in practice

- C++: uniqueness and reference counting
- Rust: affine types, unique by default, lifetimes
- Cyclone: LIFO regions, heap and stack regions
- ML Kit: viable alternative to universal GC for SML
- Clean: uniqueness types
- Idris: uniqueness and dependent types

further reading

ch. 1 (linear types),
ch. 3 (effect and region types) of
Advanced Topics in Types and Programming Languages (by Pierce)

“Linear types can change the world!”, Wadler, 1990.

“Typed memory management in a calculus of capabilities”,
Crary et. al., POPL ’99.

http://www.labri.fr/perso/renault/working/research/files/rust2016.2.pdf