CS4215 Project Presentation

Go Interpreter with Explicit-control evaluator (ECE)

Rama Aryasuta Pangestu, Yeung Man Tsung

April 19, 2024

Contents

Project Objectives

Sequential:

- ✔ Expressions
- ✔ Variable and function declarations
- ✔ Anonymous functions
- ✔ Blocks
- ✔ `if` and `for` statements
- ✔ Arrays, Slices, Structs, Strings
- ✘ `switch`, `goto`, `defer` statements
- ✘ Anonymous structs
- ✘ Using string as array (e.g. slicing)
- ✘ Pointers, Maps, Tuples
- ✘ Panic

Concurrent:

- ✔ Goroutines
- ✔ Mutex, Wait groups
- ✔ Channels
- ✔ `select` blocks
- ✘ Closing of channels

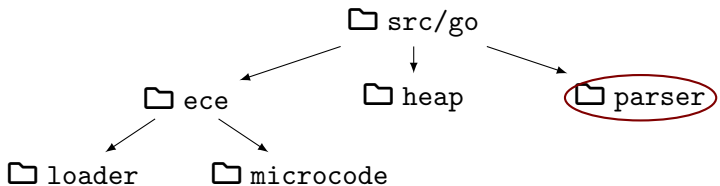
Project Objectives

Other Components:

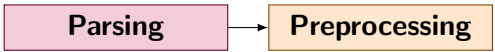
- ✔ Visualization of control, stash and current environment
- ✔ Complete memory management (including the ECE structures)
- ✔ Garbage collection using reference counting & mark and sweep
- ✔ Static type checking
- ✘ Visualization of capture frames or environment hierarchy

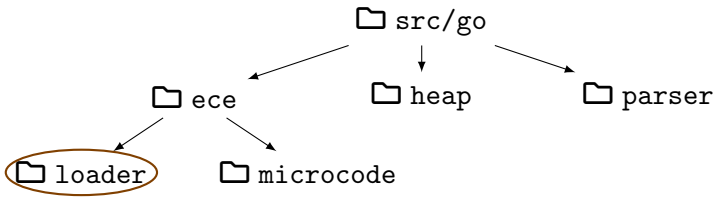
Implementation: Overview

Parsing

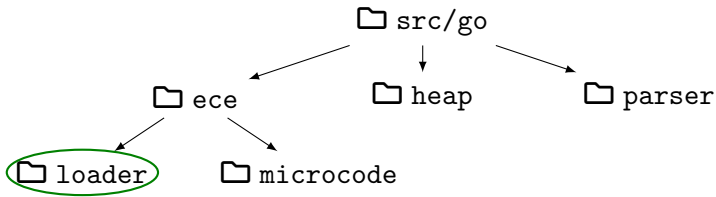


Implementation: Overview

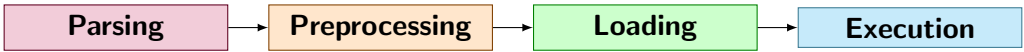


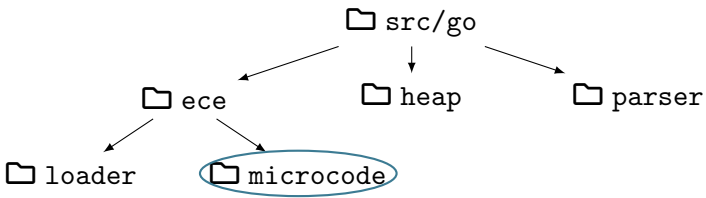


Implementation: Overview



Implementation: Overview





Four Stage Preprocessing Algorithm

Four Stage Preprocessing Algorithm

Type Checking

```
1  var x = "hello";
2  var y = x + 5; // type mismatch
```

Four Stage Preprocessing Algorithm

Type Checking

```
1  var x = "hello";
2  var y = x + 5; // type mismatch
```

Sorting

```
1  var x = y + 4; // this should come after
2  var y = 6;
```

Four Stage Preprocessing Algorithm

Type Checking ← **Sorting**

```
1  var x = "hello";
2  var y = x + 5; // type mismatch
```

```
1  var x = y + 4; // this should come after
2  var y = 6;
```

Four Stage Preprocessing Algorithm


```
1  var x = "hello";
2  var y = x + 5; // type mismatch
```

```
1  var x = y + 4; // this should come after
2  var y = 6;
```

A more complicated example:

```
1  type S struct {}
2  func (s S) Foo() int32 { return y + 1; }
3  var x = S{};
4  var y = x.Foo() + 8; // cyclic dependency: y --> S::foo() --> y
```

Four Stage Preprocessing Algorithm


```
1  var x = "hello";
2  var y = x + 5; // type mismatch
```

```
1  var x = y + 4; // this should come after
2  var y = 6;
```

A more complicated example:

```
1  type S struct {}
2  func (s S) Foo() int32 { return y + 1; }
3  var x = S{};
4  var y = x.Foo() + 8; // cyclic dependency: y --> S::foo() --> y
```

Four Stage Preprocessing Algorithm

1. **Preprocessing #1**: Compute the *captures* of all functions and global scope variable declarations. (Excluding: Methods for structs)

```
1  func main() { // captures: [y, g, fmt]
2      x := y + g();
3      go func() { // captures: [x, fmt]
4          x++
5          fmt.Println(x)
6      }()
7      z := Pair{1, 2}.first() // not captured yet
8  }
```

Four Stage Preprocessing Algorithm

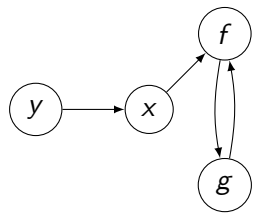
2. **Sorting #1**: Sorts the variable declarations in global scope according to the captures.

```
1  var x = f();
2  func f() int32 { return g(); }
3  func g() int32 { return f(); }
4  var y = x + 1;
```

Four Stage Preprocessing Algorithm

2. **Sorting #1**: Sorts the variable declarations in global scope according to the captures.

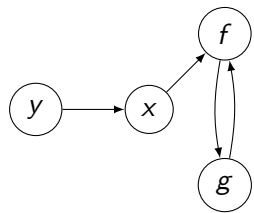
```
1  var x = f();
2  func f() int32 { return g(); }
3  func g() int32 { return f(); }
4  var y = x + 1;
```



Four Stage Preprocessing Algorithm

2. **Sorting #1**: Sorts the variable declarations in global scope according to the captures.

```
1  var x = f();
2  func f() int32 { return g(); }
3  func g() int32 { return f(); }
4  var y = x + 1;
```



Idea: Each **strongly connected component** in the graph must consist of one variable / one or more functions.

Four Stage Preprocessing Algorithm

3. **Preprocessing #2**: Performs type checking and annotate all declarations with their types. Compute captures for method calls.

```
1  func main() { // captures: [y, g, fmt, Pair::first]
2      x := y + g(); // type: int32
3      go func() { // captures: [x, fmt]
4          x++
5          fmt.Println(x)
6      }()
7      z := Pair{1, 2}.first() // type: int32
8  }
```

Four Stage Preprocessing Algorithm

3. **Preprocessing #2**: Performs type checking and annotate all declarations with their types. Compute captures for method calls.

```
1  func main() { // captures: [y, g, fmt, Pair::first]
2      x := y + g(); // type: int32
3      go func() { // captures: [x, fmt]
4          x++
5          fmt.Println(x)
6      }()
7      z := Pair{1, 2}.first() // type: int32
8  }
```

Idea: All functions and methods are already typed. There is no need to sort them in order to perform type checking.

Four Stage Preprocessing Algorithm

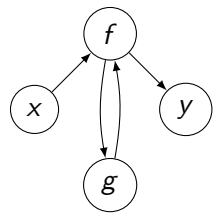
4. **Sorting #2**: Sorts all variable, function and method declarations in the global scope.

```
1  type S struct {}
2  var x = S{}.f();
3  var y = 0;
4  func (s S) f() int32 { return y + g() }
5  func g() int32 { return S{}.f() }
```

Four Stage Preprocessing Algorithm

4. **Sorting #2**: Sorts all variable, function and method declarations in the global scope.

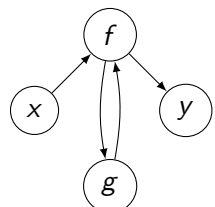
```
1  type S struct {}
2  var x = S{}.f();
3  var y = 0;
4  func (s S) f() int32 { return y + g() }
5  func g() int32 { return S{}.f() }
```



Four Stage Preprocessing Algorithm

4. **Sorting #2**: Sorts all variable, function and method declarations in the global scope.

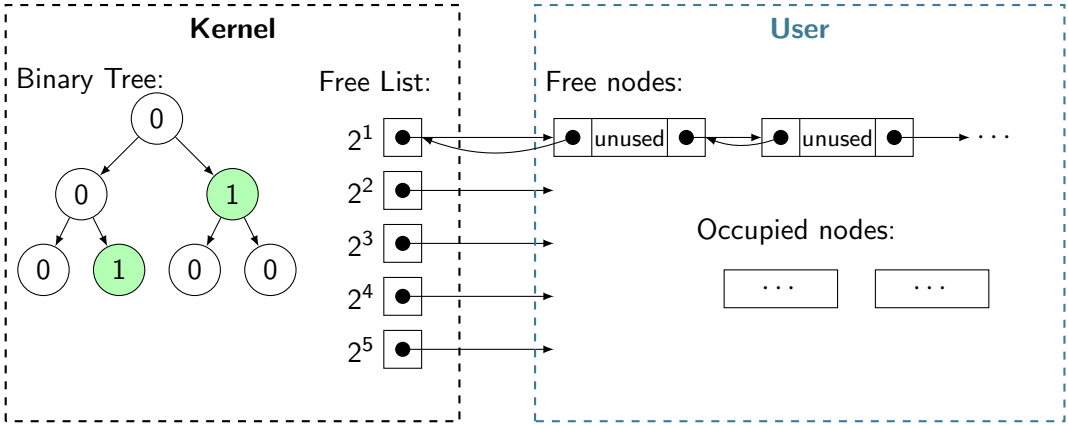
```
1  type S struct {}
2  var x = S{}.f();
3  var y = 0;
4  func (s S) f() int32 { return y + g() }
5  func g() int32 { return S{}.f() }
```



Idea: Each **strongly connected component** in the graph must consist of one variable / one or more functions and methods.

Heap

Variable-size allocations are supported by **buddy allocation**.



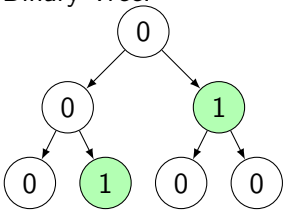
Heap

Buddy allocation: ALLOCATE

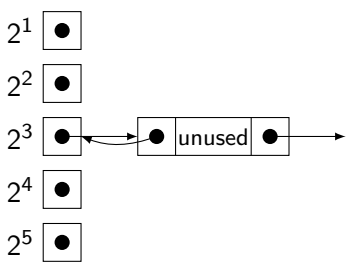
- Find a free node that is at least as large.
- Split the node to obtain the desired size.

Example: ALLOCATE(2^2)

Binary Tree:



Free List:



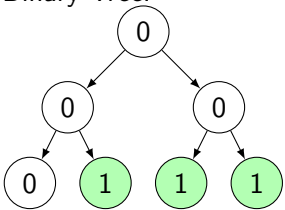
Heap

Buddy allocation: ALLOCATE

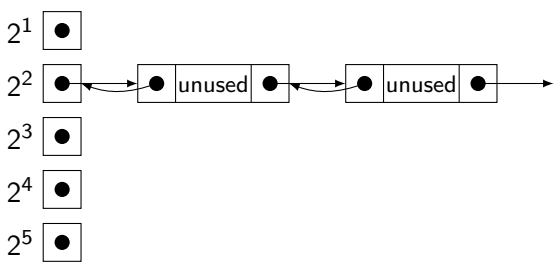
- Find a free node that is at least as large.
- Split the node to obtain the desired size.

Example: ALLOCATE(2^2)

Binary Tree:



Free List:



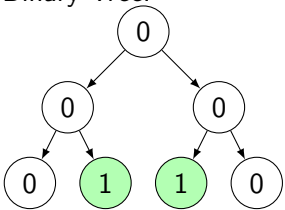
Heap

Buddy allocation: ALLOCATE

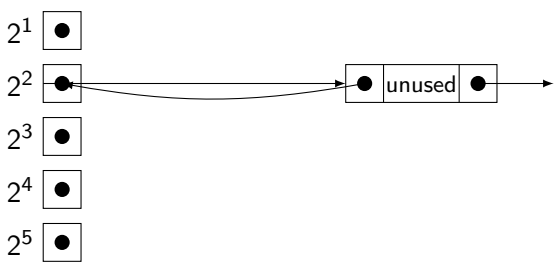
- Find a free node that is at least as large.
- Split the node to obtain the desired size.

Example: ALLOCATE(2^2)

Binary Tree:



Free List:



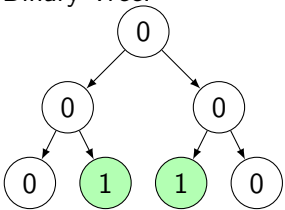
Heap

Buddy allocation: DEALLOCATE

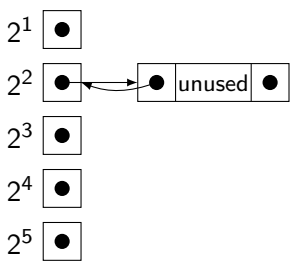
- Add the node into the free list.
- While the buddy of the node is free, merge the two free blocks.

Example: DEALLOCATE()

Binary Tree:



Free List:



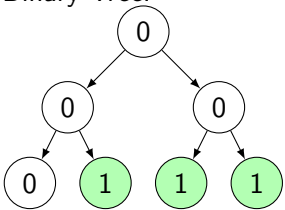
Heap

Buddy allocation: DEALLOCATE

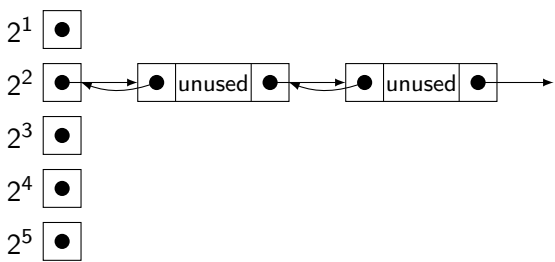
- Add the node into the free list.
- While the buddy of the node is free, merge the two free blocks.

Example: DEALLOCATE()

Binary Tree:



Free List:



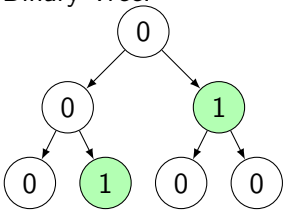
Heap

Buddy allocation: DEALLOCATE

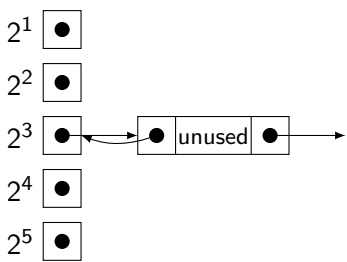
- Add the node into the free list.
- While the buddy of the node is free, merge the two free blocks.

Example: DEALLOCATE()

Binary Tree:



Free List:



Heap

Objects in the heap:

Heap

Objects in the heap:

- w words: available words that can be allocated.

Heap

Objects in the heap:

- w words: available words that can be allocated.
- $2 \log w$ words: addresses of the nodes in each "level" of the free list.

Heap

Objects in the heap:

- w words: available words that can be allocated.
- $2 \log w$ words: addresses of the nodes in each "level" of the free list.
- w bits: binary tree values, to check whether a node is in the free list.

Heap

Objects in the heap:

- w words: available words that can be allocated.
- $2 \log w$ words: addresses of the nodes in each "level" of the free list.
- w bits: binary tree values, to check whether a node is in the free list.
- special addresses (e.g., the "root addresses" for mark-and-sweep).

Heap

Objects in the heap:

- w words: available words that can be allocated.
- $2 \log w$ words: addresses of the nodes in each "level" of the free list.
- w bits: binary tree values, to check whether a node is in the free list.
- special addresses (e.g., the "root addresses" for mark-and-sweep).

Efficiency: approximately 1 bit wasted for every allocatable word $\implies \frac{32}{33} \approx 96.97\%$.

Heap

Objects in the heap:

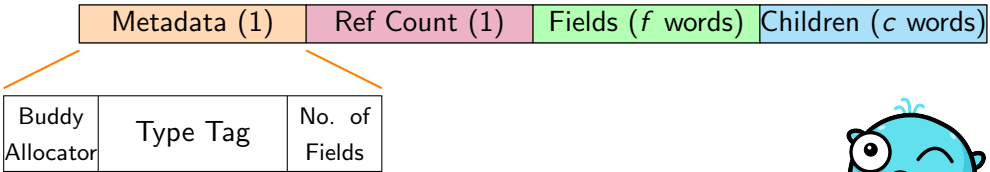
- w words: available words that can be allocated.
- $2 \log w$ words: addresses of the nodes in each "level" of the free list.
- w bits: binary tree values, to check whether a node is in the free list.
- special addresses (e.g., the "root addresses" for mark-and-sweep).

Efficiency: approximately 1 bit wasted for every allocatable word $\implies \frac{32}{33} \approx 96.97\%$.

The binary tree contains enough information to iterate all allocated nodes in $O(w)$ (e.g., for mark-and-sweep).

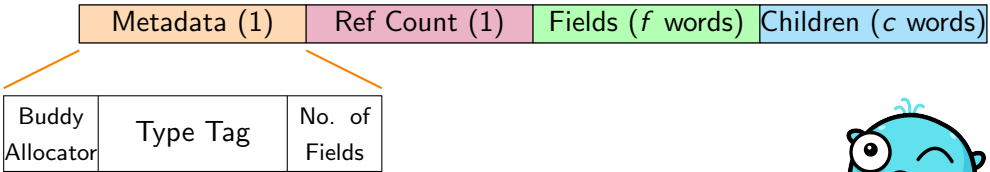
Heap

Structure of a node (word size = 4 bytes):



Heap

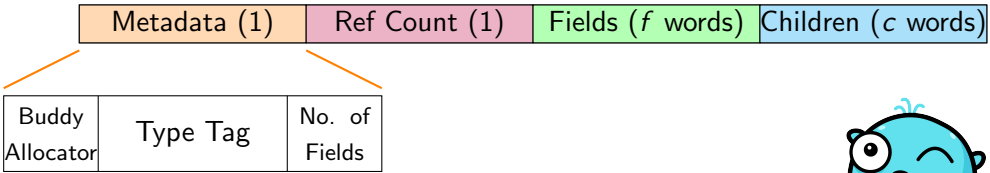
Structure of a node (word size = 4 bytes):



- Each field is a non-address additional information.

Heap

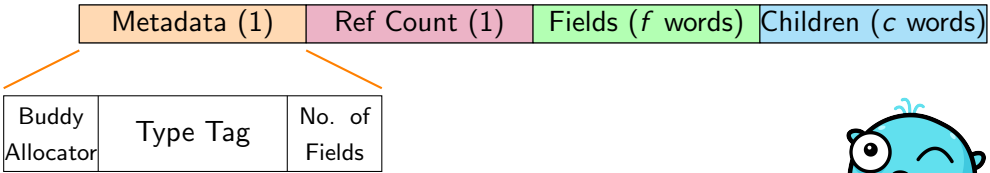
Structure of a node (word size = 4 bytes):



- Each field is a non-address additional information.
- Each child is an address pointing to another object.

Heap

Structure of a node (word size = 4 bytes):



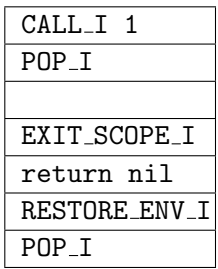
- Each field is a non-address additional information.
- Each child is an address pointing to another object.
- If an object type has children (determined by a bit in Type Tag), the first field denotes the number of children.

Context

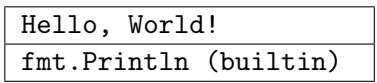
A thread consists of a control, a stash and an environment.

- Control and stash are implemented using linked lists, allowing for efficient `forking` of the threads.

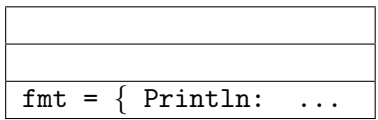
Control:



Stash:



Environment:



Environments

An environment consists of:

- A linked list of variable frames (can be pushed / popped / replaced).
- A **static** struct environment frame (types & methods).

```
1  type S struct {}
2  func (s S) foo() {}
3  func main() {
4      x := 0
5      {
6          y := 1
7          z := y + 1
8          fmt.Println(y) // here
9      }
10 }
```

Variable frames:

y = 1 z = 2
x = 0
fmt = { Println: ...

Struct frame:

S = {} METHOD.S.foo = func ...

Environments

A **frame** is a hash table, with a pointer to the parent environment frame.

Environments

A **frame** is a hash table, with a pointer to the parent environment frame.

- Hash tables use an open addressing scheme with Robin Hood linear probing.

Environments

A **frame** is a hash table, with a pointer to the parent environment frame.

- Hash tables use an open addressing scheme with Robin Hood linear probing.
- When load factor exceeds 0.6, we rehash the table with twice the size.

Environments

A **frame** is a hash table, with a pointer to the parent environment frame.

- Hash tables use an open addressing scheme with Robin Hood linear probing.
- When load factor exceeds 0.6, we rehash the table with twice the size.

Since searching through the list of frames takes O(number of frames) time, we create a cache (another hash table) at every layer to cache previously searched keys.

Environments

A **frame** is a hash table, with a pointer to the parent environment frame.

- Hash tables use an open addressing scheme with Robin Hood linear probing.
- When load factor exceeds 0.6, we rehash the table with twice the size.

Since searching through the list of frames takes O(number of frames) time, we create a cache (another hash table) at every layer to cache previously searched keys.

- The cache will contain every variable referenced in the current scope, hence the total size of all environment frame caches is bounded by O(program size).

Environments: For Loops

From Go 1.22 onwards, loop variables have **per iteration scope**.
We create a new frame per iteration with a **copy** of the loop variable.

```
1  func main() {
2      var f func() int32;
3      for i := 1; i <= 10; i++ {
4          f = func() int32 { return i; }
5          i++
6          fmt.Println(i)
7      } // output: 2, 4, 6, 8, 10
8      fmt.Println(f()) // output: 10
9  }
```

Variable frame:

i = 4
i = 3
f = func() { return i; }
fmt = { Println: ...

Demo

Environments: Function Calls

On function declaration, we create a new environment frame with the **captures**, and include an address to the frame in the closure object.

```
1   var y = 0;
2
3   func f(x int32) int32 {
4       z := 2;
5       return x + y;
6   }
7
8   func main() {
9       f(1);
10  }
```

Capture frame for `f`:

`y = 0`

Variable frame when calling `f`:

`z = 2` (body block)
`x = 1` (parameters)
`y = 0` (captures)

Demo

Concurrency

A **scheduler** is a queue of threads. The ECE pulls from the scheduler and executes one instruction on the thread.

The scheduler takes **ownership** of the threads in the queue.

Concurrency

Spawning a new thread: `go foo(1)`

Original Thread:

Control:

`foo`
1
`GO_CALL_I 1`
instructions...

Stash:

something...

Environment:

`y = ...`
`fmt = ...`

Concurrency

Spawning a new thread: `go foo(1)`

Original Thread:

Control:

GO_CALL_I 1
instructions...

Stash:

1
foo
something...

Environment:

y = ...
fmt = ...

Concurrency

Spawning a new thread: `go foo(1)`

Original Thread:

Control:

GO_CALL_I 1
instructions...

Stash:

1
foo
something...

Environment:

y = ...
fmt = ...

Spawned Thread:

Control:

CALL_I 1

Stash:

[unchanged]

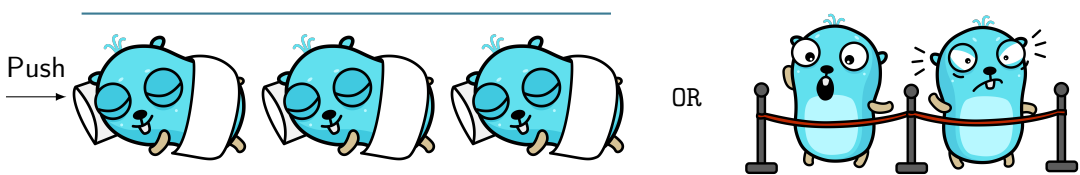
Environment:

[unchanged]

Concurrency Control: Mutex

When we call `lock()` on a mutex object in the heap, we pass ownership of the thread to it. Possible outcomes:

1. The thread is enqueued into the scheduler (non-blocking), if the mutex is currently unlocked.
2. The thread is enqueued into the **wait queue** of the mutex (blocking), if the mutex is currently locked.



An `unlock()` call moves one thread in the wait queue to the scheduler (if any).

Concurrency Control: Channels

Channels have a **send waiting queue** and a **receive waiting queue**, which is used when the buffer is full and empty respectively.

"Atomic" instructions `CHAN_SEND_I` and `CHAN_RECEIVE_I` are used to handle send and receive instructions. Similarly, we pass ownership of the thread to the channel object.

Some careful implementation is needed to handle both buffered and unbuffered channels.

Concurrency Control: Channels

When a thread sends,

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func send(t Thread, value value_t) {
8    if buffer.size() < k || waitingRecv.size() > 0 {
9      buffer.enqueue(value)
10     if waitingRecv.size() > 0 {
11       newThread := waitingRecv.dequeue()
12       newValue := buffer.dequeue()
13       newThread.Stash().Push(newValue)
14       scheduler.Wake(newThread)
15     }
16   } else {
17     waitingSend.enqueue(t)
18   }
19 }
```

Concurrency Control: Channels

When a thread sends,

- If there is a thread in the receive waiting queue, it gets unblocked (and the received value is pushed into the receiving thread's stash).

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func send(t Thread, value value_t) {
8    if buffer.size() < k || waitingRecv.size() > 0 {
9      buffer.enqueue(value)
10     if waitingRecv.size() > 0 {
11       newThread := waitingRecv.dequeue()
12       newValue := buffer.dequeue()
13       newThread.Stash().Push(newValue)
14       scheduler.Wake(newThread)
15     }
16   } else {
17     waitingSend.enqueue(t)
18   }
19 }
```

Concurrency Control: Channels

When a thread sends,

- If there is a thread in the receive waiting queue, it gets unblocked (and the received value is pushed into the receiving thread's stash).
- If the buffer is not full, it pushes the value to the buffer.

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func send(t Thread, value value_t) {
8    if buffer.size() < k || waitingRecv.size() > 0 {
9      buffer.enqueue(value)
10     if waitingRecv.size() > 0 {
11       newThread := waitingRecv.dequeue()
12       newValue := buffer.dequeue()
13       newThread.Stash().Push(newValue)
14       scheduler.Wake(newThread)
15     }
16   } else {
17     waitingSend.enqueue(t)
18   }
19 }
```

Concurrency Control: Channels

When a thread sends,

- If there is a thread in the receive waiting queue, it gets unblocked (and the received value is pushed into the receiving thread's stash).
- If the buffer is not full, it pushes the value to the buffer.
- Otherwise, the sending thread gets pushed to the send waiting queue.

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func send(t Thread, value value_t) {
8    if buffer.size() < k || waitingRecv.size() > 0 {
9      buffer.enqueue(value)
10     if waitingRecv.size() > 0 {
11       newThread := waitingRecv.dequeue()
12       newValue := buffer.dequeue()
13       newThread.Stash().Push(newValue)
14       scheduler.Wake(newThread)
15     }
16   } else {
17     waitingSend.enqueue(t)
18   }
19 }
```

Concurrency Control: Channels

When a thread `receives`,

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func recv(t Thread) {
8    if waitingSend.size() > 0 {
9      newThread := waitingSend.dequeue()
10     newValue := newThread.sendValue
11     buffer.enqueue(newValue)
12     scheduler.Wake(newThread)
13   }
14   if buffer.size() > 0 {
15     newValue := buffer.dequeue()
16     t.Stash().Push(newValue)
17   } else {
18     waitingRecv.enqueue(t)
19   }
20 }
```

Concurrency Control: Channels

When a thread `receives`,

- If the buffer is not empty, it gets a value in the buffer.

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func recv(t Thread) {
8    if waitingSend.size() > 0 {
9      newThread := waitingSend.dequeue()
10     newValue := newThread.sendValue
11     buffer.enqueue(newValue)
12     scheduler.Wake(newThread)
13   }
14   if buffer.size() > 0 {
15     newValue := buffer.dequeue()
16     t.Stash().Push(newValue)
17   } else {
18     waitingRecv.enqueue(t)
19   }
20 }
```

Concurrency Control: Channels

When a thread `receives`,

- If the buffer is not empty, it gets a value in the buffer.
- If there is a thread in the send waiting queue, it gets unblocked (and the sent value is pushed into the buffer).

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func recv(t Thread) {
8    if waitingSend.size() > 0 {
9      newThread := waitingSend.dequeue()
10     newValue := newThread.sendValue
11     buffer.enqueue(newValue)
12     scheduler.Wake(newThread)
13   }
14   if buffer.size() > 0 {
15     newValue := buffer.dequeue()
16     t.Stash().Push(newValue)
17   } else {
18     waitingRecv.enqueue(t)
19   }
20 }
```

Concurrency Control: Channels

When a thread `receives`,

- If the buffer is not empty, it gets a value in the buffer.
- If there is a thread in the send waiting queue, it gets unblocked (and the sent value is pushed into the buffer).
- Otherwise, the `receive`-ing thread gets pushed to the receive waiting queue.

```
1  type Channel struct {
2    buffer Queue  // at most size k
3    waitingSend Queue
4    waitingRecv Queue
5  }
6
7  func recv(t Thread) {
8    if waitingSend.size() > 0 {
9      newThread := waitingSend.dequeue()
10     newValue := newThread.sendValue
11     buffer.enqueue(newValue)
12     scheduler.Wake(newThread)
13   }
14   if buffer.size() > 0 {
15     newValue := buffer.dequeue()
16     t.Stash().Push(newValue)
17   } else {
18     waitingRecv.enqueue(t)
19   }
20 }
```

Concurrency Control: Select

The evaluation of a `select` block is as follows:

Concurrency Control: Select

The evaluation of a `select` block is as follows:

- We evaluate the expressions under each case (for channel send instructions).

Concurrency Control: Select

The evaluation of a `select` block is as follows:

- We evaluate the expressions under each case (for channel send instructions).
- We take a random permutation of the cases.

Concurrency Control: Select

The evaluation of a `select` block is as follows:

- We evaluate the expressions under each case (for channel send instructions).
- We take a random permutation of the cases.
- For each case in the order of our permutation, we call `try_send` or `try_receive` on the channel. We push the body of the case into the control if it succeeds.

Concurrency Control: Select

The evaluation of a `select` block is as follows:

- We evaluate the expressions under each case (for channel send instructions).
- We take a random permutation of the cases.
- For each case in the order of our permutation, we call `try_send` or `try_receive` on the channel. We push the body of the case into the control if it succeeds.
- Otherwise, we block until one of the channels can `send` or `receive`.

Concurrency Control: Select

Following `channel`'s implementation, we need to push the thread into the send or receive waiting queue of each corresponding channel.

Problems with this:

- A thread can be "woken up" multiple times.
- If a thread gets unblocked from `c1`, then goes in another waiting queue `mut`, it may get woken up from c2 and goes into the critical section before acquiring `mut`.

```
1  var mutex sync.Mutex
2  c1 := make(chan int32)
3  c2 := make(chan int32)
4
5  // some code...
6
7  select {
8      case c1 <- 10:
9          mutex.Lock()
10         // critical section
11         mutex.Unlock()
12     case c2 <- 10:
13         // some code...
14 }
```

Concurrency Control: Select

Use a layer of indirection.

```
type Waker struct {
  thread Thread
  isWokenUp bool
}

var mutex sync.Mutex
c1 := make(chan int32)
c2 := make(chan int32)

// some code...

select {
    case c1 <- 10:
        mutex.Lock()
        // critical section
        mutex.Unlock()
    case c2 <- 10:
        // some code...
}
```

Concurrency Control: Select

Use a layer of indirection.

- A `thread` is encapsulated in a `waker`.

```
type Waker struct {
  thread Thread
  isWokenUp bool
}

var mutex sync.Mutex
c1 := make(chan int32)
c2 := make(chan int32)

// some code...

select {
    case c1 <- 10:
        mutex.Lock()
        // critical section
        mutex.Unlock()
    case c2 <- 10:
        // some code...
}
```

Concurrency Control: Select

Use a layer of indirection.

- A `thread` is encapsulated in a `waker`.
- A reference to the `waker` is pushed for each channel's waiting queue. It only wakes up the `thread` the first time it is requested to by some `channel`.

```
1  type Waker struct {
2    thread Thread
3    isWokenUp bool
4  }
5
6  var mutex sync.Mutex
7  c1 := make(chan int32)
8  c2 := make(chan int32)
9
10 // some code...
11
12 select {
13     case c1 <- 10:
14         mutex.Lock()
15         // critical section
16         mutex.Unlock()
17     case c2 <- 10:
18         // some code...
19 }
```

Concurrency Control: Select

Use a layer of indirection.

- A `thread` is encapsulated in a `waker`.
- A reference to the `waker` is pushed for each channel's waiting queue. It only wakes up the `thread` the first time it is requested to by some `channel`.
- In the code, `waker1` is pushed to the waiting queue of `c1` and `c2`. When `c1` unblocks, we push `waker2` to `mut`'s waiting queue. When `c2` unblocks, it does not wake the thread currently waiting in `mut`'s waiting queue.

```
1  type Waker struct {
2    thread Thread
3    isWokenUp bool
4  }
5
6  var mutex sync.Mutex
7  c1 := make(chan int32)
8  c2 := make(chan int32)
9
10 // some code...
11
12 select {
13     case c1 <- 10:
14         mutex.Lock()
15         // critical section
16         mutex.Unlock()
17     case c2 <- 10:
18         // some code...
19 }
```

Garbage Collection

Reference Counting:

Garbage Collection

Reference Counting:

- Used to deallocate most heap objects instantly.

Garbage Collection

Reference Counting:

- Used to deallocate most heap objects instantly.
- When we call `free` on an object, it decrements the reference count and deallocates the object if the reference count reaches zero.

Garbage Collection

Reference Counting:

- Used to deallocate most heap objects instantly.
- When we call `free` on an object, it decrements the reference count and deallocates the object if the reference count reaches zero.

Unfortunately, **recursive functions** will create cyclic data structures (since it points to its own frame).

Garbage Collection

Reference Counting:

- Used to deallocate most heap objects instantly.
- When we call `free` on an object, it decrements the reference count and deallocates the object if the reference count reaches zero.

Unfortunately, **recursive functions** will create cyclic data structures (since it points to its own frame).

Implementation-wise, we use a *notion of ownership* to do reference counting. For functions such as `pop()` or `dequeue()`, we *pass the ownership* of the object to the caller (in the `microcode`). It is the caller's job to decrement the reference count of these objects.

Garbage Collection

Mark and Sweep:

Garbage Collection

Mark and Sweep:

- Used to reclaim cyclic data structures.

Garbage Collection

Mark and Sweep:

- Used to reclaim cyclic data structures.
- We maintain a list of `INTERMEDIATE_ADDRESSES` which contains the addresses to the objects possibly being used.
  - When a new object is allocated.
  - When an object is just popped from the scheduler, stash, control stack, or environment linked list.

Garbage Collection

Mark and Sweep:

- Used to reclaim cyclic data structures.
- We maintain a list of `INTERMEDIATE_ADDRESSES` which contains the addresses to the objects possibly being used.
  - When a new object is allocated.
  - When an object is just popped from the scheduler, stash, control stack, or environment linked list.
- This array is cleared after the execution of every instruction $\Rightarrow$ it does not grow over time, as each instruction only needs a constant amount of intermediate memory.

Garbage Collection

Mark and Sweep:

- Used to reclaim cyclic data structures.
- We maintain a list of `INTERMEDIATE_ADDRESSES` which contains the addresses to the objects possibly being used.
  - When a new object is allocated.
  - When an object is just popped from the scheduler, stash, control stack, or environment linked list.
- This array is cleared after the execution of every instruction $\Rightarrow$ it does not grow over time, as each instruction only needs a constant amount of intermediate memory.
- We denote some addresses as *root addresses* (e.g., the address of the `scheduler` and the address of the `main_thread`), which is treated as objects which are always in `INTERMEDIATE_ADDRESSES`. These addresses are stored in the heap's *special addresses*.

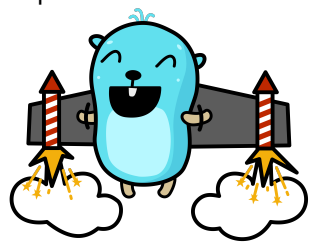
Garbage Collection

Mark and Sweep:

- Used to reclaim cyclic data structures.
- We maintain a list of `INTERMEDIATE_ADDRESSES` which contains the addresses to the objects possibly being used.
  - When a new object is allocated.
  - When an object is just popped from the scheduler, stash, control stack, or environment linked list.
- This array is cleared after the execution of every instruction $\Rightarrow$ it does not grow over time, as each instruction only needs a constant amount of intermediate memory.
- We denote some addresses as *root addresses* (e.g., the address of the `scheduler` and the address of the `main_thread`), which is treated as objects which are always in `INTERMEDIATE_ADDRESSES`. These addresses are stored in the heap's *special addresses*.
- We `mark` starting from the `INTERMEDIATE_ADDRESSES`, and `sweep` by using the heap's binary tree.

Project Outcome

- Integrated CSE machine with low-level memory management and concurrency.
- A different environment implementation using captures.
- Visualization on multi-threaded execution.
- Adopting different garbage collection schemes with minimal implementation overhead but high confidence on correctness.



Limitations

- The ECE machine is quite slow, unable to afford executing 10^5 or more iterations within a reasonable timeframe.
- Scheduling is done deterministically, thus we are unable to simulate concurrency issues such as race conditions.
- Error thrown in goroutine is propagated to main thread.