PSC 2023/24 (375AA, 9CFU)

Principles for Software Composition

Roberto Bruni
http://www.di.unipi.it/~bruni/

http://didawiki.di.unipi.it/doku.php/magistraleinformatica/psc/start

23 - Google Go
Google Go
concurrency oriented programming
Google Go

http://golang.org/
Go features

facilitate building reliable and efficient software

open source

compiled, garbage collected

functional and OO features

statically typed (light type system)

concurrent
Go principles

C, C++, Java:
too much typing (writing verbose code) and too much typing (writing explicit types) (and poor concurrency)

Python, JS:
no strict typing, no compiler issues runtime errors that should be caught statically

Google Go:
compiled, static types, type inference (and nice concurrency primitives)
Go project

designed by Ken Thompson, Rob Pike, Robert Griesemer

2007: started experimentation at Google
nov 2009: first release (more than 250 contributors)
may 2012: version 1.0 (two yearly releases since 2013)
feb 2023: version 1.20

C. Doxsey, Introducing Go (2016). Ch: 1-4, 6-7, 10
Go concurrency

any function can be executed in a separate lightweight thread

\[
\text{go } f(x)
\]

goroutines run in the same address space

package sync provides basic synchronisation primitives

programmers are encouraged NOT TO USE THEM!

\[
do \text{ not communicate by sharing memory} \\
\text{instead, share memory by communicating}
\]

use built-in high-level concurrency primitives:

channels and message passing

(inspired by process algebras)
Go channels

channels can be created and passed around

```go
var ch = make(chan int)
```
creates a channel for transmitting integers

```go
ch1 = ch
```

aliasing: ch1 and ch now refers to the same channel

```go
f(ch)
g(ch)
```

f and g share the channel ch
Directionality

channels are alway created bidirectional

\[
\text{var } \text{ch} = \text{make(chan int)}
\]

channel types can be annotated with directionality

\[
\text{var } \text{rec} \leftarrow \text{chan int}
\]

\text{rec} can only be used to receive integers

\[
\text{var } \text{snd chan} \leftarrow \text{int}
\]

\text{snd} can only be used to send integers

\[
\text{rec} = \text{ch} \\
\text{snd} = \text{ch}
\]

are valid assignments

\[
\text{rec} = \text{snd} \text{ // invalid!}
\]
Go communication

to send a value (like \texttt{ch!2}) \hspace{1cm} \texttt{ch} \leftarrow 2

to receive and store in \texttt{x} (like \texttt{ch?x}) \hspace{1cm} \texttt{x} = \leftarrow \texttt{ch}

to receive and throw the value away \hspace{1cm} \leftarrow \texttt{ch}

to close a channel (by the sender) \hspace{1cm} \texttt{close(ch)}

to check if a channel has been closed (by the receiver) \hspace{1cm} \texttt{x,ok} = \leftarrow \texttt{ch} \hspace{1cm} // \text{either value,true or 0,false}
Go sync communication

by default the communication is synchronous

BOTH send and receive are BLOCKING!

asynchronous channels can be created
by allocating a buffer of fixed size

    var ch = make(chan int, 100)

creates an asynchronous channel of size 100

receive on asynchronous channel is of course still blocking
send is blocking only if the buffer is full

no dedicated type for asynchronous channels:
buffering is a property of values not of types
Go communication
to choose between different options

```go
select {
    case x = <- ch1: { ... }
    case ch2 <- v: { ... }
    // both send and receive actions
    default: { ... }
}
```

the selection is made pseudo-randomly among enabled cases
if no case is enabled, the default option is applied
if no case is enabled, and no default option is given
the select blocks until (at least) one case is enabled
Example

non-blocking receive

```c
select {
    case x = <- ch: { ... }
    default: { ... }
}
```

receives on `x` from `ch`, if data available
otherwise proceeds
Example

wait for first among many (senders)

```haskell
select {
    case x = <- ch1: { ... }
    case x = <- ch2: { ... }
    case x = <- ch3: { ... }
}
```

receives on x from any of ch1, ch2, ch3, if data available otherwise waits
wait for first among many (receivers)

```plaintext
select {
    case ch1 <- v : { ... }
    case ch2 <- v : { ... }
    case ch3 <- v : { ... }
}
```

sends \( v \) to any of \( \text{ch1}, \text{ch2}, \text{ch3} \), if available to receive otherwise waits
Name mobility

channels can be sent over channels (like in $\pi$-calculus)

\[
\text{var } \text{mob} = \text{make(chan chan chan int)}
\]

a channel for communicating channels

\[
\text{mob } \leftarrow \text{ch}
\]

send the channel ch over mob
Name mobility: secrecy
Name mobility: secrecy
Name mobility: secrecy
Name mobility: secrecy
Name mobility: secrecy

Diagram showing interconnected nodes labeled with labels such as 'm', 'A', 'as', 'ab', 'S', 'B', 'ab', 'bs'.
Name mobility: secrecy
Name mobility: secrecy

Diagram:

- Node A with label $m$ and connections labeled as $ab$ to S and as to S.
- Node S with connections labeled as $as$ to A and $bs$ to B.
- Node B with label $bs$ and connections labeled as $ab$ to S.

Additional notation:
- $as$ to A
- $bs$ to B
- $ab$ connections between A and S, and S and B
Name mobility: secrecy
Concurrent prime sieve
Concurrent prime sieve

2, 3, 5, 7, 9, ...

Rec

ch

Gen
Concurrent prime sieve

Rec

Gen

Rec

Filt(2)

Gen

2, 3, 5, 7, 9, ...

ch

2, 3, 5, 7, 9, ...

ch

3, 5, 7, 9, ...

ch_1
Concurrent prime sieve

Conceptual diagram of the concurrent prime sieve algorithm.
1  // You can edit this code!
2  // Click here and start typing.
3  package main
4
5  import "fmt"
6
7  func main() {
8      fmt.Println("Hello, 世界")
9  }
10
11
12
13
14
15
16
17
18
19

Hello, 世界

Program exited.