Detecting Channel Blocking Errors

Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation

Oskar Haarklou Veileborg, Georgian-Vlad Saioc and Anders Møller.
Aarhus University.

Concurrency in Golang

- Golang has rich support for concurrency using goroutines and channels.
- A goroutine is a lightweight thread created by the keyword go
- Different goroutines can communicate directly through the channels.

```
func test(c chan int){
    c <- 1
}

func main(){
    c := make(chan int)
    go test(c)
    <- c
}
```

```
func test(c chan int){
    <- c
}

func main(){
    c := make(chan int, 1)
    go test(c)
    c <- 1
    c <- 2
}
```

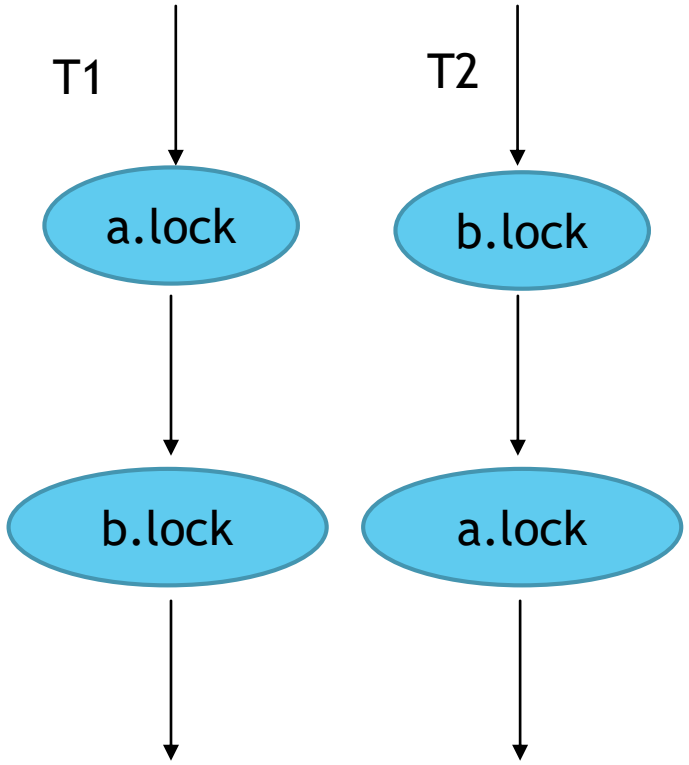
Race condition

- Given two event A and B. Suppose we have a program P can safely handle A and B.
  - Question : Does program P is safe to handle them in parallel with multiple processes ?

- If only one event A and one event B, there are two time sequences of their occurrence (A, B) and (B, A).

Race condition in multiple threads

- Deadlock



- Channel blocking

```
func main(){
    c: = make(chan int)
    go func(){
        c <- 10
    }()

    go func(){
        c <- 20
    }

    c <- 20
}
```

Sparse analysis

- Sparse analysis performs on a combined dependent graph. i.e. data-dependency graph and flow-dependency graph.

```
a = a + b
y = a + c
x = a + d
```

```
if(a > 1){
    x = y + z
}
```

Dependent channels

- C1: Given a channel c is dependent on c' if an operation on c that may unblock another operation on c is intra-procedurally reachable from a blocking operation c'.

```
a, b := make(chan int), make(chan int)
go func() {
    <-b
    <-a
}()
a <- 1
b <- 2
```

Dependent channels

- C2: Given a channel c is dependent on c' if they are used in different cases of the same select statement.

```
a, b := make(chan int), make(chan int)
select {
    case <-a:
    case <-b:
}
```

Dependent channels

- C3: Given a channel c is dependent on c' if c' might carry c as a payload.

```
a := make(chan chan int, 1)
go func() {
    b := make(chan int)
    a <- b
    b <- 3
}()
<-<-a
```

Construct program fragment.

- We use $\mathcal{P}(c)$ denote a set of dependent channels of c.
- A program fragment is a set of functions.
- Construct a program fragment $\mathcal{F}(c)$ as follows.
  - (1) If a function contains channels related to one channel in $\mathcal{P}(c)$, then the function in $\mathcal{F}(c)$.
  - $\mathcal{F}(c)$ includes all ancestors in the call graph, up to the dominator of those functions in (1).
  - The dominator is the entry of $\mathcal{F}(c)$.

Side-effect analysis

- Some the target function of call site in the program fragment may be not recursive in the program fragment.
- But these function may have side-effects for the variables in program fragments.

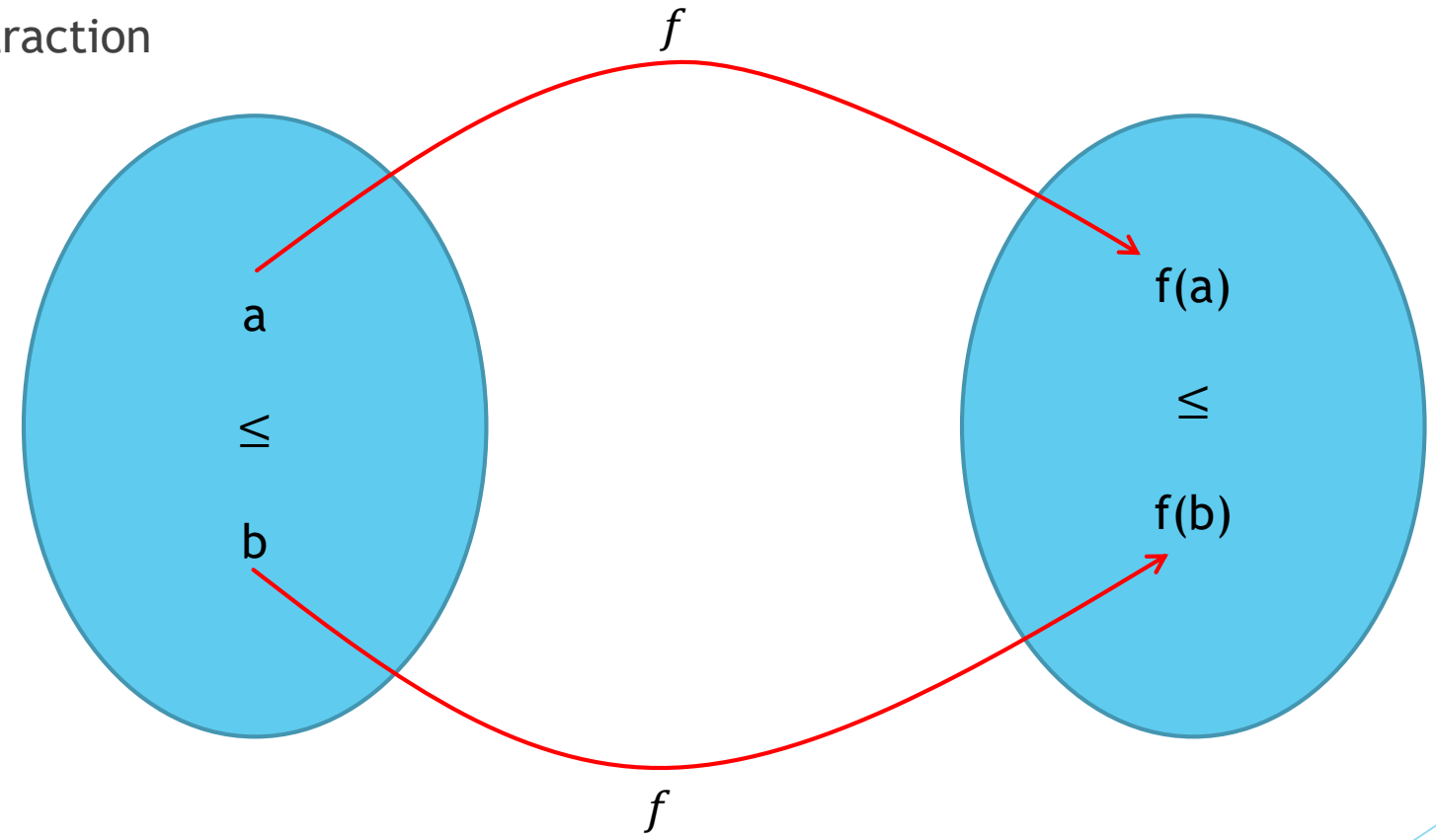
```
type S struct { ch chan int; val int; flag bool }
func entry () {
    s := S{ch: make(chan int, 1), val: 10, flag: false} 86
    init(&s)
    s.ch <- s.val
}

func init(s *S) {
    if s.flag {
        s.val = 0
    }
}
```

Abstract interpretation

- Collecting semantics : the result of static analysis
- Lattice theory
  - If a value is described by both l_1 and l_2, then we can obtain the more precise information $l_1 \wedge l_2$.
  - If a value is describe by either l_1 or l_2, the most precise that we can infer is $l_1 \vee l_2$.
- Smaller = more precise, bigger = more safe.
- Monotone function is good, but it is not enough sometime.
  - Underlying set of lattice is large
  - Or ascending chain condition (ACC) or descending chain condition (DCC) do not be hold.

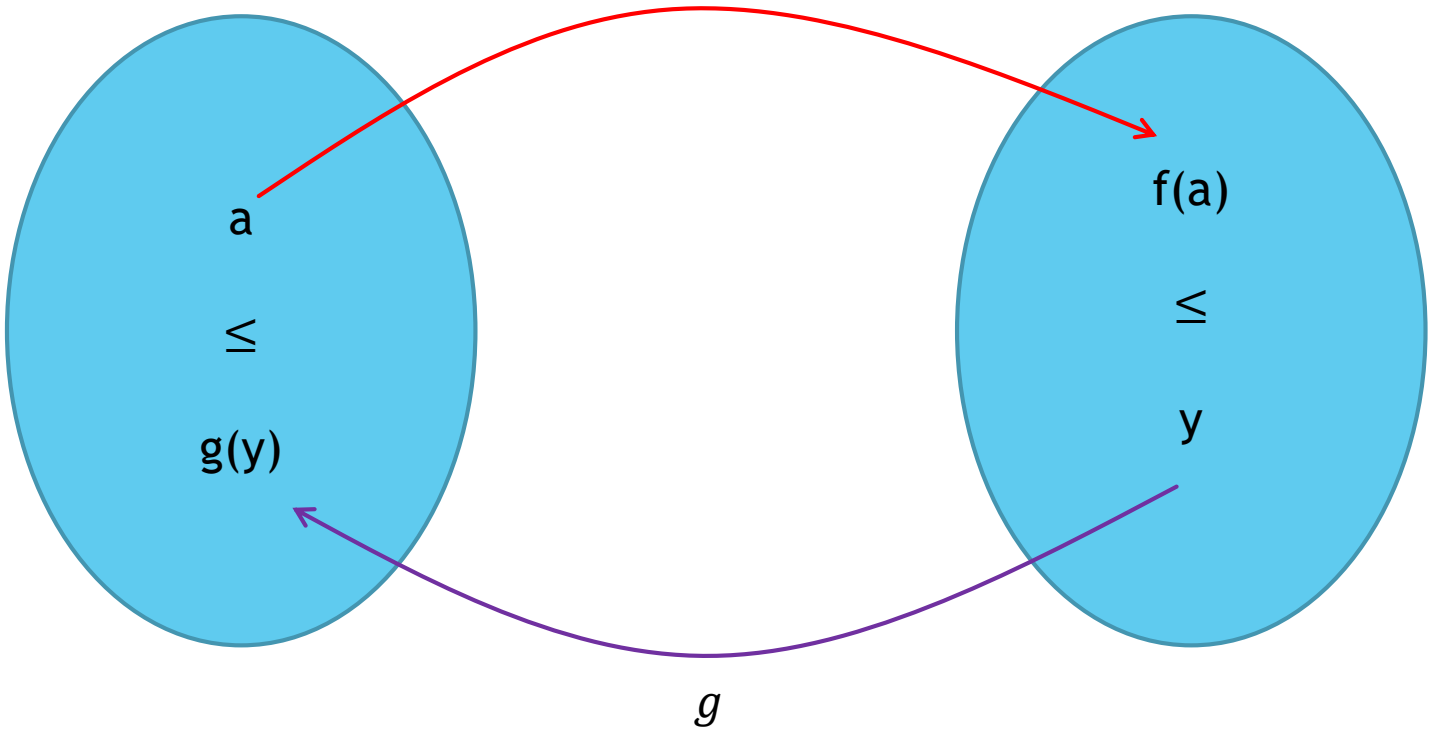
Abstract interpretation

- Abstraction



Abstract interpretation

- Safety (can be induced by a Galois f connection)



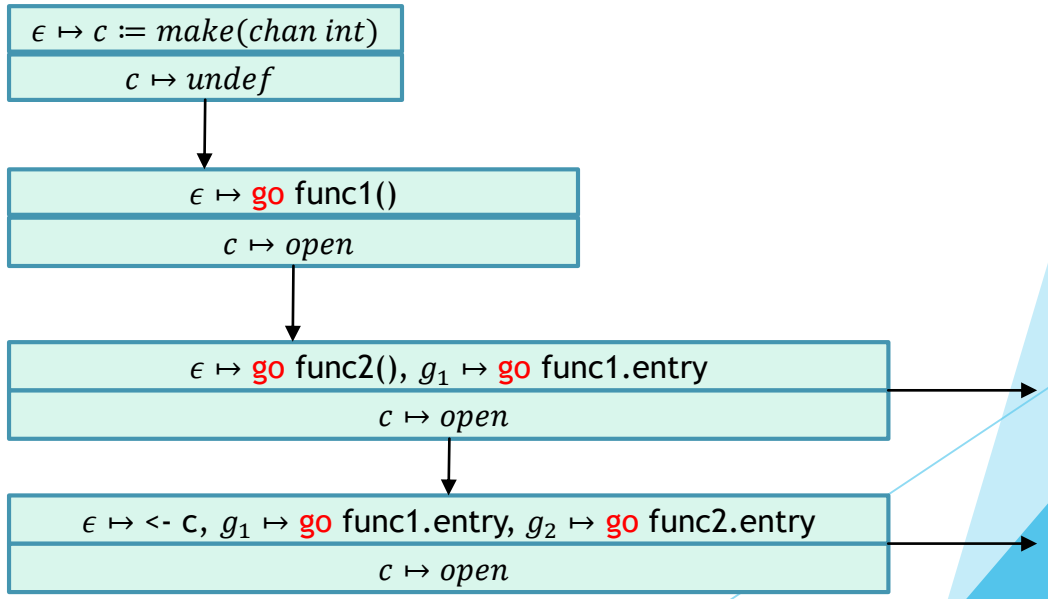
Some important domains

- Channel status: OPEN, CLOSED, undefined, and unknown.
- Channel capacity: it is actually fixed constant, because once a channel is created, its capacity (buffer size) cannot be changed.
- Channel current buffer size: an interval lattice bounded by capacity.
- Channel payload : a set of possible channels.

Modelling thread schedule

- Now we have program fragment $\mathcal{F}(c)$ with a entry and analysis method by abstract interpretation.
- It is necessary to give a thread schedule to detect concurrency errors.

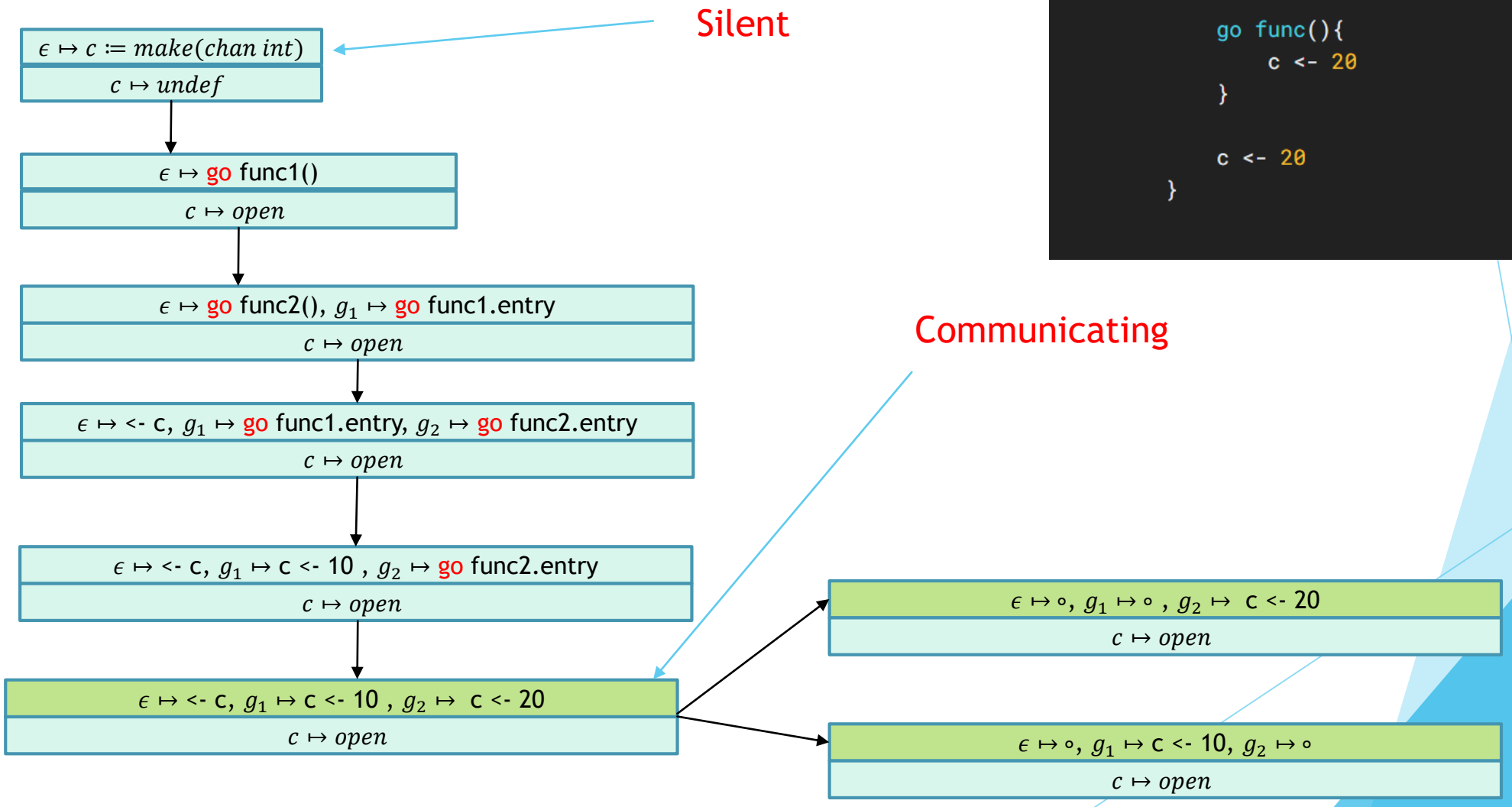
```
func main(){
    c: = make(chan int)
    go func(){
        c <- 10
    }()

    go func(){
        c <- 20
    }

    c <- 20
}
```



Modelling thread schedule

```
func main(){
    c: = make(chan int)
    go func(){
        c <- 10
    }()

    go func(){
        c <- 20
    }

    c <- 20
}
```

Full graph



Some details

- How to deal multiple threads from same go ? i.e., multiple same function call and loops.

```
func main(){
    c := make(chan int)
    for i := 0; i < 2; i++ {
        go func(){
            c <- i
        }
    }

    <-c
}
```

```
17:50:13 main.go:378: Entry 1 of 1: simple-exa
mples/sync-two-goros-race.main
17:50:13 main.go:488: 1 primitives outside GOR
OOT reachable from simple-examples/sync-two-go
ros-race.main

17:50:13 main.go:509: Found PSet 1 of 1 :
{ t1 = make chan int 0:int at /home/maple/go/s
rc/github.com/cs-au-dk/goat/examples/src/simpl
e-examples/sync-two-goros-race/main.go:4:12 (s
imple-examples/sync-two-goros-race.main) }

17:50:13 main.go:513: Using simple-examples/sy
nc-two-goros-race.main as entrypoint
17:50:13 main.go:533: Superlocation graph size
: 7
17:50:13 main.go:545: SA completed in 365.282µ
s
17:50:13 main.go:550: No blocking bugs detecte
d
17:50:13 main.go:599: Completed runs: 1, skipp
ed runs: 0, aborted runs: 0
➜ family goat git:(master) ✗
```

```
package main

func main() {
        ch := make(chan int)
        for i:= 0; i < 2; i++{
                go func() {
                        ch <- 10 //@ analy
sis(false)
                }()
        }

        //go func() {
        //      ch <- 20 //@ analysis(fals
e)
        //}()

        <-ch //@ analysis(true)
}
~
~
~
~
<l] 16L, 212C                 1,1           All
[0] 0:zsh*                          "family" 17:50 07-3月-23
```

Evaluation

- Implement a open source tool named Goat.
- Goat achieves an acceptable true positive ratio of more than 50% (99/179, 157/239).
- Goat is quicker than SMT based techniques.
- The efficiency of central treatment of loops and thread scheduling designs.

Mode	TP	FP	Aborts	Timeouts
Normal	123	53	76%	69
Sound loops	109	46	79%	66
Fine-grained scheduler	113	47	73%	780

Table 5: Importance of design choices.

Thanks!
Any questions?