Repetition Lecture

Einar Broch Johnsen, S. Lizeth Tapia Tarifa, Eduard Kamburjan, Juliane Päßler

November 17, 2023

University of Oslo

Lecture 1 - Basics

Summary

- Shared Memory: variables accessed from multiple threads
- Interleaving semantics: handling multiple threads by interleaving their statements
- Problems with analysis: state space explosion
- Basic terminology: atomicity, synchronization, mutual exclusion
- Data races, interference, AMO

Interaction Between Parallel Processes

To organize interactions, we use *synchronization*

Synchronization

Synchronization *restricts* the possible interleavings of parallel processes
to avoid unwanted behavior and enforce wanted behavior.

Definition (Atomic)

An operation is atomic if it cannot be subdivided into smaller operations.

- We can ignore concurrency inside atomic operations as they cannot be interleaved
- Assignments `x := e` are *not* atomic

- Increasing *atomicity* and *mutual exclusion* (Mutex) to introduce *critical sections* which can *not* be executed concurrently
- *Condition synchronization* enforces that processes must wait for a specific condition to be satisfied before execution can continue.

Properties

Definition (Invariant)

An *invariant* is a property of program states, that holds for all reachable states of a program.

- *Invariant* (adj): constant, unchanging
- Prototypical safety property
- Appropriate for non-terminating systems (does not require a final state)
- *All* reachable states often too strong

Kinds of Invariants

- Strong invariant: Holds for all reachable states
- Weak invariant: Holds for all states where an atomic block starts or ends
- Loop invariant: Holds at the start and end of a loop body
- Global invariant: Reasons about state of many processes
- Local invariant: Reasons about state of one process

Critical Sections

To enforce atomicity, we have a special construct in the language : <S> performs S atomically

Use of Critical Sections

- When the processes interfere: *synchronization* to restrict the possible interleavings
- Synchronization gives coarser grained atomic operations ("atomic blocks")
- Combines operations into an *atomic lock* where the process shall not be interrupted

Characteristics of Atomic Operations

- Internal states are *not visible* to other processes.
- Variables *cannot* be changed underway by other processes.

Await

```
int x:=0; co <x:=x+1> || <x:=x−1> oc {x=0}
```

Lecture 2 - Java

Summary

- Threads and runnables
- Weak memory: breaks interleaving semantics through memory optimization
- **volatile** and **synchronized**

Threads

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

`Start` vs `run`

- `Thread.start()` starts a new *concurrent* thread
- `Runnable.run()` just executes the code *sequentially*
- `Thread.start()` calls `Runnable.run()` internally
- Calling `Runnable.run()` directly rarely makes sense

Synchronization

- Synchronized blocks can be used outside methods with explicit lock
- Any object can be a lock, synchronized methods have **this** as the lock

Java

```
public class C(){
  int I = 0;
  void method(Object lock, boolean left){
    synchronized(lock){
      if(left) I++ else I--; }}}

public class D(){
  synchronized void method(){ ... }
  void method(){ synchronized(this) {...} } }
```

Weak Memory

Weak Memory Models can lead to very unintuitive results in concurrent settings

```
int x,y; //default 0
```

```
x  := 1;  //shared variable
r1 := y;  //register
print r1;
```

```
y  := 1;  //shared variable
r2 := x;
print r2;
```

- If the read of `x` in the second thread is reordered, then `0,0` is possible
- This output cannot be explained by reasoning about interleavings
- If the language does not require variables to be initialized, we get *out-of-thin-air* values. Then, even `12,13` is a possible output.

Weak Memory

Sequential Consistency

Most weak memory models guarantee *sequential consistency*: *If there is no data race, then the observable behavior of the program is as if under a strong memory model*.

- "No data race" may be a very strong restriction and lead to unnecessary synchronization
- The term *observable behavior* depends on the programming language
- We need more fine-grained control – **volatile** forbids reordering of accesses to this field

Lecture 3 - Locks and Barriers

Summary

- Implementing $\langle$**await** $e;\ s\rangle$
- Critical sections
- Test-and-set, spin-lock for lock variables
- Contention and Fairness
- Barriers: synchronization at the same point for n threads

General patterns for critical sections

- inside the CS we have operations on shared variables.
- Access to the CS must then be protected to prevent interference.
- Coarse-grained pattern for n uniform processes repeatably executing some critical section

Await

```
process p[i=1 to n] {
  while (true) {
    CSentry          # entry protocol to CS
    CS
    CSexit           # exit protocol from CS
    non-CS
  }
}
```

- *Assumption*: A process which enters the CS will eventually leave it.

⇒ *Programming advice*: be aware of exceptions inside CS!

Critical sections using “locks”

Await

```
bool lock := false;
process [i=1 to n] {
  while (true) {
    < await (!lock) lock := true >;
    CS;
    lock := false;
    non–CS
  }
}
```

Safety Properties

- Mutex
- Absence of deadlock and absence of unnecessary waiting

Can we remove the angle brackets $<$ ... $>$?

Critical section with TS and spin-lock

Await

```
bool lock := false;

process p [i=1 to n] {
  while (true) {
    while (TS(lock)) {skip};        # entry protocol
    CS
    lock := false;                  # exit protocol
  }
}
```

- TS(lock): set to true, return original value
- Safety: Mutex, absence of deadlock and of unnecessary delay.
- Strong fairness is needed to guarantee eventual entry for a process
- Problematic memory access pattern: lock as a hotspot

Reducing Writes

Test, Test and Set

Test, Test and Set (TTAS) reduces the number of writes by introducing more reads in the entry protocol.

Await

```
bool lock = false;
process p[i = 1 to n] {
  while (true) {
    while (lock) {skip};    # additional spin lock
    while (TS(lock)) { while (lock) {skip} };
    CS;
    lock := false;
  }
}
```

Conditional Atomic Sections

Implementation of < **await** (B) S;> :

Await

```
CSentry;
while (!B) {CSexit ; CSentry};
S;
CSexit;
```

Fairness notions

Possible status changes

- Disabled → enabled
- Enabled → disabled

In our language, only conditional atomic segments can have status changes

Different forms of fairness for different forms of statements

1. For statements that are always enabled
2. For those that once they become enable, they *stay enabled*
3. For those whose enabledness shows “on-off” behavior

Fairness

Definition (Unconditional fairness)

A scheduling strategy is *unconditionally fair* if each enabled unconditional atomic action, will eventually be chosen.

Definition (Weak fairness)

A scheduling strategy is *weakly fair* if

- unconditionally fair
- every conditional atomic action will eventually be chosen, assuming that the condition becomes true and *remains true* until the action is executed.

Fairness

Definition (Strongly fair scheduling strategy)

- unconditionally fair and
- each conditional atomic action will eventually be chosen, if the condition is true infinitely often.

Await

```
bool x := true; y := false;
co while (x) {y:=true; y:=false} || < await(y) x:=false > oc
```

Lecture 4 - Semaphores

Summary

- Semaphores: built-in synchronization mechanisms
- Special variable with 2 operations, cannot be accessed directly

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

The Semaphore Operations: *P* and *V*

- **P:** (Passeren) Wait for signal – want to *pass*
  Wait until value is greater than zero, and *decrease* value by one
- **V:** (Vrijgeven) Signal an event – *release*
  Increase the value by one

- Today, libraries and sys-calls prefer other names: up/down, wait/signal, acquire/release
- Different flavors of semaphores: binary vs. counting
- Most common: mutex as a synonym for binary semaphores

Syntax and Semantics

Declaration

- sem s;   default initial value is zero
- sem s := 1;
- sem s[4] := ([4] 1);

Operations and Semantics

P-operation P(s)

$\langle \text{await } (s > 0) s := s - 1 \rangle$

V-operation V(s)

$\langle s := s + 1 \rangle$

Processes waiting on a semaphore are woken up by the op. system.

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like `if (s = 1) then ... else` is *forbidden*!

Kinds of semaphores

General semaphore: Possible values: *all non-negative integers*

Binary semaphore: Possible values: 0 and 1

Example: Mutual Exclusion (critical section)

Mutex implemented by a *binary semaphore*

Await

```
sem mutex := 1;
process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    # critical section
    V(mutex);
    # noncritical section
  }
}
```

- The semaphore is *initially 1*
- Always P before V $\rightarrow$ (used as) binary semaphore

Example: Barrier Synchronization

Semaphores may be used for *signaling events*

Await

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); # reach barrier
    P(arrive2); # wait for other
    ...
}
process Worker2 {
    ...
    V(arrive2); # reach barrier
    P(arrive1); # wait for other
    ...
}
```

Split Binary Semaphores

Split binary semaphore

A *set* of semaphores, whose *sum* ≤ 1

Mutex by split binary semaphores

- Initialization: *one* of the semaphores =1, all others = 0
- Discipline: all processes call P on a semaphore, *before* calling V on (*another*) semaphore

$\Rightarrow$ Code between the P and the V

  - All semaphores = 0
  - Code executed *in mutex*

Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```

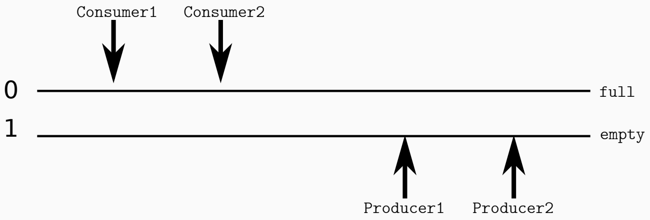
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

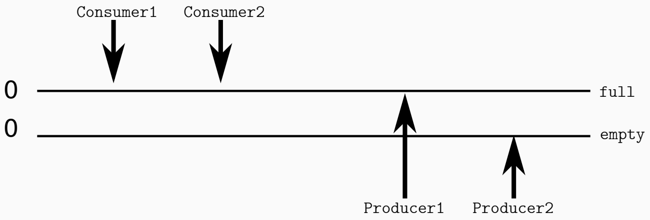
```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



Example: Producer/Consumer with Split Binary Semaphores

Await
```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await
```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



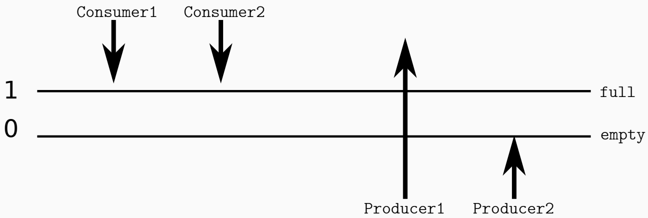
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



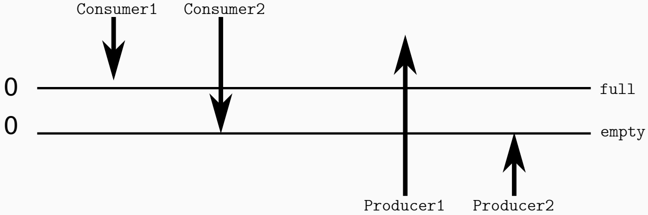
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



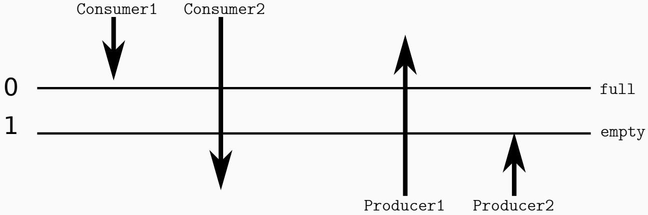
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {
  while (true) {
    P(empty);
    buff := data;
    V(full);
  }
}
```

Await

```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```



Lecture 5 - Monitors

Summary

- Special synchronization mechanism: program module encapsulating some data
- Fields accessible only through its procedures
- Synchronization through condition variables

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:

Intuitively, a monitor is an abstract data type with built-in synchronization.

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:
Intuitively, a monitor is an abstract data type with built-in synchronization.

State of a Monitor

- Contains variables that describe the *state*
- Variables can be *changed only* through the available procedures

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:
Intuitively, a monitor is an abstract data type with built-in synchronization.

State of a Monitor

- Contains variables that describe the *state*
- Variables can be *changed only* through the available procedures

Synchronization of a Monitor

Implicit mutual exclusion: at most one procedure may be active at a time for a monitor

- A procedure has guaranteed mutex access to the data in the monitor
- Two procedures in the same monitor are never executed concurrently

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:
Intuitively, a monitor is an abstract data type with built-in synchronization.

Cooperative Scheduling: procedures coordinate their monitor access

- Condition synchronization blocks a process until a particular condition holds.
- Condition synchronization is expressed by *condition variables*
- Monitors can be implemented using locks or semaphores

Monitor Usage

- Process $=$ active $\Leftrightarrow$ Monitor: $=$ passive/re-active
- A procedure is *active*, if a statement in the procedure is executed by some process

Monitor-Based Concurrency

- *All* shared variables: inside the monitor
- Processes *communicate* by calling monitor procedures
- Processes do not need to know all the implementation details

- Only the visible effects of public procedures are important
- Implementation can be changed, if visible effects remains
- Monitors and processes can be developed relatively independent of each other

$\Rightarrow$ Monitors make it *easier to understand* and develop parallel programs

Syntax & Semantics

Await

```
monitor name {
 monitor variables
 ## monitor invariant
 initialization code
 procedures
}
```

- Only the procedure names are visible from outside the monitor:

call `name.procedure(arguments)`

- Statements *inside* a monitor: *no* access to variables *outside* the monitor
- Statements *outside* a monitor: *no* access to variables *inside* the monitor
- **Monitor variables:** *initialized* before the monitor is used
- **Monitor invariant:** describes a condition on the inner state
- The monitor invariant can be analyzed by sequential reasoning inside the monitor

Condition Variables

- Monitors contain a *special* type of variables: **cond**
- Condition variables are used for synchronization/to *delay* processes
- Each *condition variable* is associated with a *wait condition*
- The "*value*" of a condition variable: *queue* of delayed processes
- This *value* is not directly accessible by programmer
- Instead, it is *manipulated* by *special operations*

```
cond cv;         # declares a condition variable cv
empty(cv);       # asks if the queue on cv is empty
wait(cv);        # causes process to wait in the cv queue
signal(cv);      # wakes up a process in the queue to cv
signal_all(cv);  # wakes up all processes in the cv queue
```

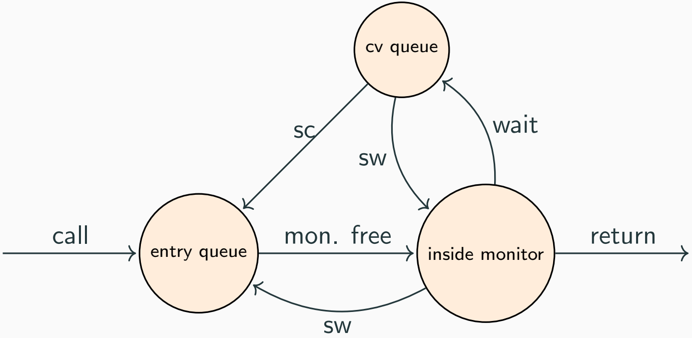
Signaling Disciplines (1)

- Statement `signal(cv)` has the following effect
  - *Empty queue:* no effect
  - *Otherwise:* the *process* at the head of the queue to `cv` is *woken up*
- A process executes `signal(cv)` while it is active
  — how to activate the next process?

Signaling Disciplines

- *Signal and Wait (SW):* the signaler waits, and the signaled process gets to execute immediately
- *Signal and Continue (SC):* the signaler continues, and the signaled process executes later

Signaling Disciplines (2)



Note: *Two kinds of queues*: entry queue and condition variable queue

Note: The figure is *schematic* and combines the "transitions" of signal-and-wait and signal-and-continue in a single diagram. The corresponding transition, here labeled sw and sc are the state changes caused by being *signaled* in the corresponding discipline.

Lecture 6 and 7 - Message Passing Concurrency

Summary

- Channels: synchronous and asynchronous channels
- Actors: monitors with asynchronous communication (and more!)
- Call-backs: (composable) futures, promises, channels, identities
- Language design: “colored” functions with async/await

Concurrent vs. distributed programming

Shared-Memory Systems

- Processors share one memory
- Processors communicate via reading and writing of shared variables

Concurrent programming provides primitives to synchronize over memory

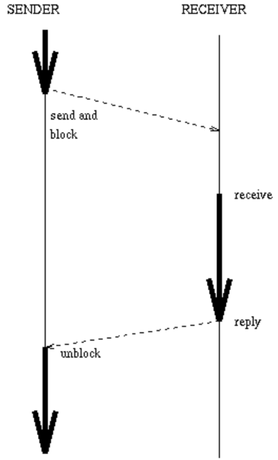
Distributed Systems

- Memory is distributed: processes cannot share variables/memory locations
- Processes communicate by sending and receiving *messages* via e.g., shared *channels*,
- or (in future lectures): communication via *RPC* and *rendezvous*

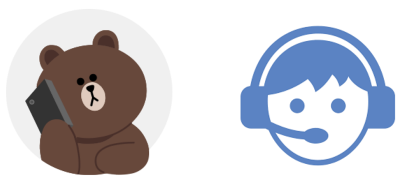
Distributed programming provides primitives to communicate

- Some concepts from distributed systems are also useful abstractions for shared memory
- Abstractions can be decoded to different primitives, e.g., channels can shared-memory
- Also: mixed shared-distributed systems

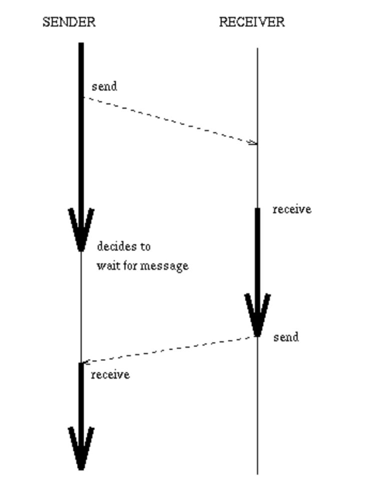
Synchronous message passing - high level concept



Synchronous message passing



Asynchronous message passing - high level concept



Asynchronous message passing

Actors

Fundamental idea: Decouple communication and control.

Capabilities of Actors

An actor reacts to incoming messages to

- change its state,
- send a finite number of messages to other actors, and
- create a finite number of new actors.

Intuition

We can think of an actor as an object that can only communicate asynchronously, but some actor models can also pattern match over its message queue of incoming messages.

Actors

Actors

- Recipients of messages are identified by name (no channels).
- An actor can only communicate with actors that it knows.
- An actor can obtain names from messages that it receives,
  or because it has created the actor

The actor model is characterized by

- inherent concurrency among actors
- dynamic creation of actors,
- inclusion of actor names in messages, and
- interaction only through direct asynchronous message passing with no restriction on message arrival order.
- *message servers* might be implemented by matching messages from the queue to procedures

Example: Erlang-style Actors - Matching Messages
Publish and Subscribe Server

```
runServer(Subs) ->
  receive
    {sub, from} -> runServer(Subs + from);  % subscribe
    {publish, value} ->                      % publish
                for(id in Subs) id!{value}, % broadcast value
                runServer(Subs);
    _ -> runServer(Subs); % ignore other messages

Server { % publish and subscribe server
  start() -> spawn(fun() -> runServer([])).} % start the server

Client { % send requests to the server
  start() -> Server!{sub, self}, Server!{publish,10}.}
```

Futures and Promises

Futures.

- A future is a handle for the caller of a process that will contain the result value once computed
- Most commonly: return value of a process

Java

```
Future<Int> f = service.submit(() -> { return 1;});
...
Int = f.get();
```

Promises.

- What if the value will be computed somewhere else?
- A *promise* is a future which is not clear who computes it

Promises

A promise:

- May be eventually completed (but maybe by somebody else)
- Can be completed only once
- Deadlock/starvation occurs if it is never completed

Java calls promises *CompletableFutures*:

Java

```
CompletableFuture<Integer> f = new CompletableFuture<>();
service.submit(() -> { f.complete(1); return null;});
...
Int = f.get();
```

Composition Futures/Promises

Logically related Futures/Promises scattered in the code.

Java

```
CompletableFuture<Integer> f1
  = CompletableFuture.supplyAsync(() -> 1);
   ...
CompletableFuture<Integer> f2
  = CompletableFuture.supplyAsync(() -> f1.get + 1);
//Connecting Futures/Promises  (composition)
CompletableFuture<Integer> f
  = CompletableFuture.supplyAsync(() -> 1)
                             .thenApply((res) -> res + 1);
```

Very similar patterns are common in web development with JavaScript

Active Objects: actors + object-orientation

- Each object runs one thread and each method call spawns a *task*
- Thread is responsible to schedule tasks in some order
- Waiting on future suspends the task, not the thread!
- Reading blocks task and thread – no other task can run

ABS

```
class Diner(IWaiter w) implements IDiner {
    Unit eat(Dish d) {
        Fut<Meal> fm = w!order(d); // place order with waiter
        await fm?; // while waiting do something else
        Meal m = fm.get; // receive meal
        Fut<Unit> fc = this!consume(m);
        Fut<Unit> fp = w!pay(this, d); // eating, paying in some order
        await fc? & fp?; // eaten and paid — ready to leave!
    }
    Unit takeCall(){ ... }
    ...
}
```

Lecture 8 - Go

Go Concurrency

Go's concurrency mantra

"Don't communicate by sharing memory, share memory by communicating!"

- Go does have shared memory via global variables, heap memory etc.
- But you are supposed to only send references – getting a reference transfers ownership, i.e., the permission to write/read it

Go's primitives

- Goroutines – lightweight threads
  - Own call stack, small stack memory (2KB initially), handled by go runtime
  - Very cheap context switch
  - First-class constructs of language
- Channels
  - Synchronous, Typed
  - Communication between (lightweight) threads
  - Main means of synchronization

Goroutines

3 ways to call a function

- `f(x)` – ordinary (synchronous) function call, where f is a defined function or a functional definition
- `go (x)` – called as an asynchronous process, i.e. go-routine
- `defer f(x)` – the call is delayed until the end of this process

Channels in Go

- Channels provide a way to send messages form one go routine to another.
- Channels are created with **make**
- The arrow operator (<-) is used both to signify the direction of a channel and to send or receive data over a channel

Go

```go
func m(){
  chl := make(chan float64)
  go writef(chl);  go readf(chl)
}
func writef(ch chan<- float64) {
    ch <- 0.5 }
func readf(ch <-chan float64){
    v := <-ch }
```

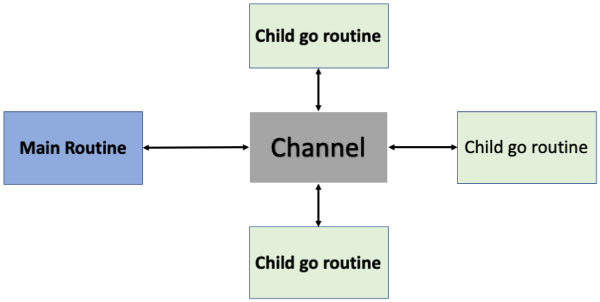
Waiting for Go routines to finish

Go

```
func main() {
    var wg sync.WaitGroup
    var i int = -1
    var file string
    for i, file = range os.Args[1:] {
        wg.Add(1) //add before async. call!
        go func() { //anon. function
            compress(file)
            wg.Done()}()
    }
    wg.Wait()
    fmt.Printf("compressed %d files \n", i+1)
}
```

Channels in Go

- Channels are bidirectional, synchronous and typed
- Careful which routine is reading and which is writing
- Type support to enforce that



Channel operations

- Send and receive
- Create channels
- Close a channel

Channel operations

- Send and receive
- Create channels
- Close a channel

Go

```go
func m() {
  ch := make(chan int, 2) //async channel
  ch <- 1 //does not block!
  ch <- 2
  fmt.Println(<-ch)
  fmt.Println(<-ch)
}
```

Channel operations

- Send and receive
- Create channels
- Close a channel

Go

```go
func m() {
  ch := make(chan int, 1) //async channel
  ch <- 1
  ch <- 2 //deadlock
  fmt.Println(<-ch)
  fmt.Println(<-ch)
}
```

Channel operations

- Send and receive
- Create channels
- Close a channel

Go

```go
func m() {
  ch := make(chan int) //sync channel
  go write(ch)
  for {
      i, ok := <-ch
      if(!ok) break
      fmt.Println(i) } }

func write(ch chan<- int) {
  ch <- 1; ch <- 2; close(ch) }
```

Channel operations

- Send and receive
- Create channels
- Close a channel

Go

```
func m() {
  ch := make(chan int)
  go write(ch)
  <-ch
  <-ch } // error

func write(ch chan<- int) {
  ch <- 1; close(ch) }
```

Lecture 9 - General Types

Summary

A typing discipline consists of

- A type syntax
- A subtyping relation
- A typing environment
- A type judgement
- A set of type rules (the type system itself)
- A notion of type soundness

Data and Behavioral Types

- A data type is an abstraction over the contents of memory
  - Can it be interpreted as a member of a set? E.g., integers
  - Are certain operations *defined* on it? E.g., + or method lookup
- A behavioral type is an abstraction over *allowed* operations

Environment and Judgment

Type Environment

A type environment Γ is a partial map from variables to types.

- Notation to access the type of a variable $\texttt{v}$ in environment Γ: $\Gamma(\texttt{v})$
- Example notation for an environment with two integer variables $\texttt{v}, \texttt{w}$: $\{\texttt{v} \mapsto \texttt{Int}, \texttt{w} \mapsto \texttt{Int}\}$
- Notation for updating the environment: $\Gamma[\texttt{x} \mapsto \texttt{T}]$
- Notation if a variable has no assigned type: $\Gamma(\texttt{x}) = \bot$

Type Judgment

To express that statement $\texttt{s}$ is well-typed with type T in environment Γ.

$$\Gamma \vdash \mathsf{e} : \texttt{T}$$

Type Soundness

Type soundness expresses that if the initial program is well-typed, then we do not get stuck, i.e., if we terminate, then *successfully*.

- Three intermediate lemmas (error states are not well-types, subject reduction, progress)
- Note that we do not ensures termination
- Main thinking point for later: are deadlocked states successfully terminated?

Subject Reduction

If a well-typed expression can be execute, then the result is well-typed

$$\forall s, s', \Gamma.\ \big((\Gamma \vdash s : \texttt{Unit} \wedge s \rightsquigarrow s') \to \exists \Gamma'.\ \Gamma' \vdash s' : \texttt{Unit}\big)$$

Progress

If a statement is well-typed, but not successfully terminated (i.e., **skip** or **return**), then it can make a step

$$\forall s.\ \big((\Gamma \vdash s : \texttt{Unit} \wedge \neg \mathsf{term}(s) \to \exists s'.\ s \rightsquigarrow s'\big)$$

Types for Channels

Typing Writing

$$\frac{\Gamma \vdash \texttt{e} : \texttt{chan T} \qquad \Gamma \vdash \texttt{e}' : \texttt{T}' \qquad \texttt{T}' <: \texttt{T}}{\Gamma \vdash \texttt{e} \ <\!- \ \texttt{e}' : \texttt{Unit}}$$

- First premise types channel
- Second premise types sent value
- Third premise connects via subtyping

Typing Reading

$$\frac{\Gamma \vdash \texttt{e} : \texttt{chan T'} \qquad \texttt{T'} <: \texttt{T}}{\Gamma \vdash \ <\!- \ \texttt{e} : \texttt{T}}$$

Input/Output Modes

- How to enforce that one thread reads and one writes?
- Idea: use modes to encode read or write capabilities
- Use subtyping and weakening to split and restrict capabilities

```
func main() {
  chn := make(chan!? int) //!?
  go read(chn)            //!?
  //weaken chn to chan! int
  chn <- v //<- chn would be illegal
}
func read(c chan? int) int { //forgets ! mode
  return <-c //c <- 1 would be illegal
}
```

Lecture 10 - Linear and Usage Types

Summary

- Linear types, affine types, usage types
- Unrestricted environment
- Splitting the environment (and type)
- Different ways to enforce order: in rules or in specification

Comparison: Session Types (ST) and other Channel Types

Type System	Form	Split	Order	Guarantee	Specification	Expressiveness
Data Types	chan int	–	–	Data Safety	Minimal	No communication patterns
Modes	$\texttt{chan}_{?!}$ int	Implicit in rules	Implicit in rules arbitrary often	–	Minimal Only interfaces	Distinguishes reader from writer
Linear Types	$\texttt{chan}_{?1,!1}$ int	Implicit in rules	Implicit in rules once	No DL on single channels	Minimal Only interfaces	Single-use channels, distinguishes reader from writer
Usage Types	$\texttt{chan}_{!.?.0+?.!.0}$ int	Explicit in spec.	Explicit in spec.	–	Considerate No consistency checking	Simple protocols, more than 2 participants
Binary ST	chan !int.?string.0	Implicit at declaration	Explicit in spec.	No DL on single channels	Medium effort Consistency checked	Complex protocols with branching between 2 participants
Multi-Party ST	chan $p \to q$: int.0	Extra mechanism	Explicit in spec.	No DL on single channels	Considerate Consistency checked	Complex protocols with branching between n participants

Linear Types: Defining Splitting

Typing Environment

A typing environment Γ can be split into two environments $\Gamma^1 + \Gamma^2$ by

- Having all variables with non-channel types in both Γ^1 and Γ^2.
- For each x with channel type we have $\Gamma(x) = \Gamma^1(x) + \Gamma^2(x)$, where

$$\mathtt{chan}_{?\mathtt{n}^1,!\mathtt{m}^1}\ T + \mathtt{chan}_{?\mathtt{n}^2,!\mathtt{m}^2}\ T = \mathtt{chan}_{?\mathtt{n}^1+\mathtt{n}^2,!\mathtt{m}^1+\mathtt{m}^2}\ T$$

- $\mathtt{chan}_{?1,!1}\ \mathrm{T} = \mathtt{chan}_{?0,!1}\ \mathrm{T} + \mathtt{chan}_{?1,!0}\ \mathrm{T}$
- $\mathtt{chan}_{?1,!1}\ \mathrm{T} = \mathtt{chan}_{?1,!1}\ \mathrm{T} + \mathtt{chan}_{?0,!0}\ \mathrm{T}$

$$\{\mathrm{n} \mapsto \mathtt{Int}, \mathrm{c} \mapsto \mathtt{chan}_{?0,!1}\ \mathtt{Int}\} =$$
$$\{\mathrm{n} \mapsto \mathtt{Int}, \mathrm{c} \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{Int}\} + \{\mathrm{n} \mapsto \mathtt{Int}, \mathrm{c} \mapsto \mathtt{chan}_{?0,!1}\ \mathtt{Int}\}$$

Linear Types: Defining Complete Use

Literals and Termination

- Γ is unrestricted if all contained channels have $n = 0$ and $m = 0$. We write $\mathsf{un}(\Gamma)$.
- All literals only type check in a unrestricted environment
- First, sub-system only for for expressions

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathit{true} : \mathtt{Bool}}\ \text{L-true} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash n : \mathtt{Int}}\ \text{L-int}$$

$$\frac{\mathsf{un}(\Gamma) \qquad \Gamma(\mathtt{v}) = T}{\Gamma \vdash \mathtt{v} : T}\ \text{L-var}$$

$$\frac{\overline{\mathsf{un}(\Gamma)}}{\{c \mapsto \mathbf{chan}_{?0,!0}\} \vdash 1 : \mathtt{Int}}$$

$$\{c \mapsto \mathbf{chan}_{?1,!0}\} \vdash 1 : \mathtt{Int}$$

Type Soundness – Enforce Parallelism

Writing

- Check that we can write *but not read* c now
- Remove write capability and split the environment into two parts
- One (Γ_1) records the write capability and the capabilities afterwards
- One (Γ_2) record the capabilities of the evaluated expression
- The first must allow one write
- The second must allow no read – otherwise one can type `c <- c`
- Also prohibits sequential self-locks `c <- 1; <- c`

$$\frac{\Gamma[\mathtt{c} \mapsto \mathbf{chan}_{?\mathbf{0},!0}\ T] = \Gamma_1 + \Gamma_2 \qquad \Gamma(\mathtt{c}) = \mathbf{chan}_{?\mathbf{0},!1}\ T \qquad \Gamma_1 \vdash \mathtt{s} : \mathtt{Unit} \qquad \Gamma_2 \vdash \mathtt{e} : T}{\Gamma \vdash \mathtt{c}\ <-\ \mathtt{e};\ \mathtt{s} : \mathtt{Unit}}$$

Usage Types by Example

Go

```
func main(){
  global = 0
  lock := make(chan<!.?.0 + ?.!.?.!.0 + ?.!.?.!.0> int)
  finish := make(chan<?.?.0 + !.0 + !.0> int)

  go dual(1, lock, finish)
  go dual(2, lock, finish)
  lock <- 0
  <-finish
  <-finish
}
```

Example

- Let $\Gamma = \{\texttt{lock} \mapsto \texttt{chan}_{!.?.0+?.!.?.!.0+?.!.?.!.0}\ \texttt{Int},\ \texttt{finish} \mapsto \texttt{chan}_{?.?.0+!.0+!.0}\ \texttt{Int},\ \texttt{global} \mapsto \texttt{Int}$
- Let $\Gamma_1 = \{\texttt{lock} \mapsto \texttt{chan}_{!.?.0+?.!.?.!.0}\ \texttt{Int},\ \texttt{finish} \mapsto \texttt{chan}_{?.?.0+!.0}\ \texttt{Int},\ \texttt{global} \mapsto \texttt{Int}$
- Let $\Gamma_2 = \{\texttt{lock} \mapsto \texttt{chan}_{?.!.?.!.0}\ \texttt{Int},\ \texttt{finish} \mapsto \texttt{chan}_{!.0}\ \texttt{Int},\ \texttt{global} \mapsto \texttt{Int}$

$$\dfrac{\dfrac{\vdots}{\Gamma_1 \vdash \texttt{s} : \texttt{Unit}} \qquad \dfrac{\vdots}{\Gamma_2 \vdash \texttt{dual}(1,\ \texttt{lock},\ \texttt{finish}) : \texttt{Unit}}}{\Gamma = \textbf{go}\ \texttt{dual}(1,\ \texttt{lock},\ \texttt{finish});\ \texttt{s} : \texttt{Unit}}$$

Lecture 11 - Session Types

Summary

- Session: a channel used for a single communication
- Different data types communicated
- Treating two endpoints of a channel with different types and variables
- Duality to ensure endpoints match
- active choice (I chose) vs. passive choice (I react)
- Multi-party settings: duality generalizes to projection

Requirements for Session Types

Session

A *session* is a sequence of related interactions between ≥ 2 parties over a certain time frame.

- Idea: a channel is only used for a single session
- A linear type describes a session with a single interaction
- A usage types describes a complex session and distributes interactions using $+$

Requirements for a Type System for Sessions

- Specify precisely possible orders of operations as protocols
- Clarify roles in sessions
- Must be able to handle branching in protocols
- Must be able to send different data types during protocol

Two Views on Channels

Establishing a Session

Creating a channel results in two values, for two endpoint

$$(\mathtt{x}, \mathtt{y}) \ := \ \mathtt{make}(\mathtt{chan}\ \mathtt{T}_1, \mathtt{chan}\ \mathtt{T}_2)$$

- The values of `x`, `y` have the "same" channel.

Binary Session Types

- Make sure types T_1, T_2 match using *duality*
- Channel is used for one session described by T_1, T_2 and completed on termination

Binary Session Types

Type Syntax

- Data type of sent values is now part of protocol/session type
- Additional difference to usage types: no $+$

$T ::=$	S	Session Type
	$\mid \texttt{chan}\ T$	Channel Type
	$\mid D$	Data types
$S ::=$	$!T.S$	Send
	$\mid ?T.S$	Receive
	$\mid 0$	Termination
	...	(next page)

Session to tend an integer and get some Boolean answer: `chan !int.?bool.0`

Binary Session Types

Type Syntax

$$
\begin{aligned}
S ::= \;& \ldots && \text{(previous page)} \\
\mid \;& \oplus\{l_1 : S_1, \ldots, l_n : s_n\} && \text{Internal choice} \\
\mid \;& \&\{l_1 : S_1, \ldots, l_n : s_n\} && \text{External choice}
\end{aligned}
$$

(repetition $*$ can be added if needed)

- Intuition: The party using $\oplus$ decides on branch and send the label
- Intuition: The party using & receives the label and continues with the corresponding branch

Binary Session Types

Duality

- Ensures that both parties communicating over a channel have a *symmetric* or dual view.
- Given a binary session type, we can syntactically construct its dual.
- Alternatively: Given two binary session types, we chan check whether they are duals

$$\begin{aligned}
\overline{0} &= 0 \\
\overline{!T.S} &= ?T.\overline{S} \\
\overline{?T.S} &= !T.\overline{S} \\
\overline{\&\{l_1 : S_1, \ldots, l_n : S_n\}} &= \oplus\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\} \\
\overline{\oplus\{l_1 : S_1, \ldots, l_n : S_n\}} &= \&\{l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\}
\end{aligned}$$

Subtyping and Type Soundness

Subtyping

Subtyping has same idea of duality as the type.

- Internal choice can have more branches
  Intuition: active choice to never take these branches.
- External choice can have less branches
  Intuition: these branches are never chosen anyway.

$$\oplus\{l_i : S_i\}_{i \in I} <: \oplus\{l_i : S'_i\}_{i \in I'} \qquad \text{iff } I \supseteq I' \land \forall i \in I.\ S_i <: S'_i$$

$$\&\{l_i : S_i\}_{i \in I} <: \&\{l_i : S'_i\}_{i \in I'} \qquad \text{iff } I \subseteq I' \land \forall i \in I'.\ S_i <: S'_i$$

Multi-Party Session Types

Type Syntax

Unify all constructs into one type expressing that p sends a label l_i together with a data value of type T_i to q, and the communication continues as S_i.

$$S ::= 0 \mid p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\}$$

We omit the outermost parentheses if $n = 1$.

$$\text{Alice} \to \text{Bob} : \left\{ l_1(\texttt{int}) : \text{Bob} \to \text{Carol} : \left\{ \begin{array}{ll} l_2(\texttt{int}) : & \text{Carol} \to \text{Alice} : l_4(\texttt{int}).0 \\ l_3(\texttt{int}) : & \text{Carol} \to \text{Alice} : l_4(\texttt{int}).0 \end{array} \right\} \right\}$$

Multi-Party Session Types

Local Types

Two actions, which are the unification of internal choice and sending, and the unification of external choice and receiving.

$$
\begin{aligned}
L ::= & \, 0 \\
& \mid \&\{p_1?l_1(T_1).L_1, \ldots, p_n?l_n(T_n).L_n\} \\
& \mid \oplus\{q_1!l_1(T_1).L_1, \ldots, q_n!l_n(T_n).L_n\}
\end{aligned}
$$

$$
\begin{aligned}
L_{\text{Alice}} &= \text{Bob}!l_1(\texttt{int}).\text{Carol}?l_4(\texttt{int}).0 \\
L_{\text{Bob}} &= \text{Alice}?l_1(\texttt{int}).\oplus\begin{Bmatrix} \text{Carol}!l_2(\texttt{int}).0 \\ \text{Carol}!l_3(\texttt{int}).0 \end{Bmatrix} \\
L_{\text{Carol}} &= \&\begin{Bmatrix} \text{Bob}?l_2(\texttt{int}).\text{Alice}!l_4(\texttt{int}).0 \\ \text{Bob}?l_3(\texttt{int}).\text{Alice}!l_4(\texttt{int}).0 \end{Bmatrix}
\end{aligned}
$$

Multi-Party Session Types

- Duality is generalized to *projection*: generate a local type for each role from a global type
- When projecting on receiver q, turn $p \to q$ into a ?
- When projecting on sender p, turn $p \to q$ into a !
- When projecting on any one else, each branch must be the same – this party is not communicated which branch is taken

Projection

Generate local type $L_p = G \upharpoonright p$ from global type G and role p

$$0 \upharpoonright p = 0$$

$$p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\}_{\upharpoonright} p = \oplus\{q!l_1(T_1).(S_1 \upharpoonright p), \ldots, q!l_1(T_1).(S_1 \upharpoonright p)\}$$

$$p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\} \upharpoonright q = \&\{p?l_1(T_1).(S_1 \upharpoonright q), \ldots, p?l_n(T_n).(S_n \upharpoonright q)\}$$

$$p \to q : \{l_1(T_1) : S_1, \ldots, l_n(T_n) : S_n\} \upharpoonright r = L \text{ where } \forall i.\ L = S_i \upharpoonright r$$

Lecture 12 - Rust

Summary

- Ownership: Affinity types for *copying*
- Only one writing pointer per memory cell
- Automatic deallocation, data race freedom
- Owner vs. reference
- Lifetime of owner is lifetime of value

Connecting Syntax and Semantics

Resource Allocation Is Initialization (RAII)

- Memory management for local (=stack) instances
- Memory used by class allocated by constructor
- Memory deallocated by destructor
- No explicit deallocation needed, destructor called upon leaving the stack scope

```
class C { public int* p;
        C() { p = new int[4]; }
        ~C() { delete[] data; } }

void f() {
        C c();
        c.f(); }
```

Ownership

- Reassignment of ownership *(as in let b = a)* is a move
- Affinity is considered with respect to moves
- Once ownership has been given away, a variable can no longer be used
- *a* is “used up” and therefore unusable
- Values with copy trait and literals are not moved, but copied

Rust

```
fn main() {
let a = vec![1, 2, 3];                    // a vector
let b = a;                                // move: ‘a‘ can no longer be used
println!("{0} {1}", a[0], b[0]);   //error : borrow of moved value : 'a'
}
```

Ownership

- Reassignment of ownership *(as in let b = a)* is a move
- Affinity is considered with respect to moves
- Once ownership has been given away, a variable can no longer be used
- *a* is "used up" and therefore unusable
- Values with copy trait and literals are not moved, but copied

Rust

```
fn main() {
let a = 1:
let b = a;                    // not a move: 'a' is copied!
println!("{0} {1}", a, b); //works
}
```

Passing Ownership

Passing a value also passes ownership of the value

Rust

```
fn make_vec() -> Vec<i32> {
  let mut vec = Vec::new();
  vec.push(1);
  vec  // transfer ownership back to the caller
  }
fn use_vec() {
  let vec = make_vec(); // take ownership of the vector
  print_vec(vec);       // pass ownership to print_vec
}
fn print_vec(vec: Vec<i32>) { // vec is owned by print_vec
  for i in vec.iter()
    println!("{}", i)
 }  // now, vec is deallocated
```

Passing Ownership

Rust

```
fn use_vec() {
  let vec = make_vec();   // take ownership of vector
  print_vec(vec);         // pass ownership to print_vec

  for i in vec.iter()     // ERROR: continue using vec
    println!("{}", i * 2)
}
```

- Ownership is not transferred again by *print_vec*, vec it destroyed here.
- Trying to use the *vec* again gives an *error*
- More than just “discipline”: the vector has already been *deallocated* at this point!

Lifetime

- Deallocation is handled by *lifetimes*
- Value, references and variables all have lifetimes
- A reference/variable has a lifetime from until it goes out-of-scope.
- A value has a lifetime until its owener goes out-of-scope
- References are not owners: Reference must have shorter lifetimes than their value

Lifetime

- Deallocation is handled by *lifetimes*
- Value, references and variables all have lifetimes
- A reference/variable has a lifetime from until it goes out-of-scope.
- A value has a lifetime until its owener goes out-of-scope
- References are not owners: Reference must have shorter lifetimes than their value

Rust

```
fn main() {
  let rf;                       //----+ Lifetime of ref
  {                             //    |
    let vec = vec![1, 2, 3];    //--+ | Lifetime of vec and value
    rf = &vec;                  // |  | ref borrows read access
  }                             //--+ | vec goes out-of-scope, value deallocated
  println!("{}", (*rf)[0]);     //    | access invalid!
}                               //----+
```

Lifetime

- Deallocation is handled by *lifetimes*
- Value, references and variables all have lifetimes
- A reference/variable has a lifetime from until it goes out-of-scope.
- A value has a lifetime until its owener goes out-of-scope
- References are not owners: Reference must have shorter lifetimes than their value

Rust

```
fn main() {
    let vec = vec![1, 2, 3];       //----+
    let rf = &vec;                 //-+  |
    println!("{}", (*rf)[0]);      // |  |
}                                  //-+--+
```

Referencing in Rust

A reference to a value cannot outlive the owner

Rust

```
let v = vec![1 , 2];
let x=&v[0];
let v2=v;        // Owner changes from v to v2!
let y = *x + 1 // ERROR — x refers to v, but v is not an owner!
```

A value can have one mutable reference or many immutable references

Rust

```
let mut  v = vec![1 , 2];
let x=&v[0];                 // immutable borrow here
Vec::push (&mut v , 3);      // ERROR: mutable borrow here
let y = *x +1;               // removing this line fixes the example!
```

Exam

Languages

Languages for Answers

- Await and monitor languages for shared memory
- "Erlang" for actors, Go for channels and types, Rust
- All needed typing rules will be included in the exam sheet, typing derivations can be entered in ASCII

Languages

Languages for Answers

- Await and monitor languages for shared memory
- "Erlang" for actors, Go for channels and types, Rust
- All needed typing rules will be included in the exam sheet, typing derivations can be entered in ASCII

```
----------------------
{ x -> int }(x) = int
-----------------------
{ x -> int } |- x : int         (2)
-------------------------------------
{ x -> int } |- int v = x; s : Unit
```

```
...
{ x -> int, v -> int } |- s : Unit
-----------------------------------
                (2)
```

Languages

Languages for Answers

- Await and monitor languages for shared memory
- "Erlang" for actors, Go for channels and types, Rust
- All needed typing rules will be included in the exam sheet, typing derivations can be entered in ASCII

- Other languages (Java, ABS, . . .) must be understood, can occur in questions
- For Rust, you can expect $<$10 lines of code
- Slight syntax derivations are allowed, but do not mix languages

Expectations (incomplete! list of possible question formats)

- Explain a concept
  "What is a linear channel?"
- Explain the differences between concepts
  "What are the differences and commonalities between a monitor and an actor?"
- Given a scenario, implement it in concurrency model X
- Given a scenario, model it in type system X
- Given some code, does it have property Y (**describe why, why not**)
  "Can this code deadlock?"
- Given some code, does it have property Y under assumption Z (**describe why, why not**)
  "Can this code deadlock under the signal-and-continue discipline?"
- Compute properties of types: splits, duals, projections, subtypes
- Given some code, give a type in System X for the channel to make it type-safe
- Given some code, does it type check? If yes, give a type derivation, otherwise

Typing Exercises

Give binary session types for S, T so the program is well-typed. Labels are prefixed with t_.

```
func f(i int){
 (c1, c2) := make(chan S, S_Dual)
 go f(c2);
 if( i > 0 ) {
     c1 <- l_big
     d := <-c1
     d <- i
 } else {
     c1 <- l_small
     c1 <- i
     d := <-c1
     d <- 0-i
 }
}
```

```
func g(c chan S){
 (d1, d2) := make(chan T, T_Dual)
 switch <- c{
   case l_big:
     c <- d1
     print(<-d2)
   case l_small:
     v := <-c
     c <- d1
     print(<-d2 + v)
 }
}
```

Typing Exercises

Give binary session types for S, T so the program is well-typed. Labels are prefixed with `t_`.

```
func f(i int){
 (c1, c2) := make(chan +{l_big: ?(chan !int.0).0,
                         l_small: !int.?(chan !int.0).0, S_Dual)
 go f(c2);
 if( i > 0 ) { c1 <- l_big; d := <-c1; d <- i}
 else { c1 <- l_small; c1 <- i; d := <-c1; d <- 0-i }}

func g(c chan ...){
 (d1, d2) := make(chan !int.0, T_Dual)
 switch <-c {
   case l_big: c <- d1; print(<-d2)
   case l_small: v := <-c; c <- d1; print(<-d2 + v) }}
```

Typing Exercises

Show that the following is not well-typed and annotate the environment in every line when after type checking this statement. Explain which property of usage types is violated.

```
func f(){
 c := make(chan<!.?.0+?.!.0> int);
 i := 0;
 go g(c);
 c <- i;
 <- c;
 skip;
}
```

```
func g(c chan<?.!.0> int){
 v := <- c
 if( v <= 0 ){
     c <- 0;
     skip;
 } else {
     skip;
 }
 skip
}
```

Typing Exercises

```
func f(){
 c := make(...); // {c -> chan<!.?.0+?.!.0> int}
 i := 0;         // {c -> chan<!.?.0+?.!.0> int, i -> int}
 go g(c);        //   {c -> chan<!.?.0> int, i -> int}
                 // + {c -> chan<?.!.0> int, i -> int}
 c <- i;         // {c -> chan<?.0> int, i -> int}
 <- c;           // {c -> chan<0> int, i -> int}
 skip            // {c -> chan<0> int, i -> int}
}
```

Typing Exercises

```
func g(c chan<?.!.0>){
 v := <-c;          // c -> chan<!.0> int , v -> int}
 if( v <= 0 ){      // c -> chan<!.0> int , v -> int} (then)
     c <- 0;        // c -> chan<0> int , v -> int}
     skip;          // c -> chan<0> int , v -> int}
 } else {           // c -> chan<!.0> int , v -> int} (else)
     skip;          // c -> chan<!.0> int , v -> int}
 }
}
```

Usage types require that the communication is performed until the end, but g does not always send back.

Typing Exercises

Is the following Rust code well-typed? Annotate the lifetimes of x, y, z, the owner of the x in every line and argue whether type checking succeeds.

Rust

```
fn f(mut z : Vec<i32>){ ... }
fn g(q : &Vec<i32>){ ... }
fn main() {
    let mut x = vec![1,2,3];
    g(&x);
    let y = x;
    f(y);
    g(&y);
}
```

Typing Exercises

Is the following Rust code well-typed? Annotate the lifetimes of x, y, owner of the created verctor in main in every line and argue whether type checking succeeds.

Rust

```
fn f(mut z : Vec<i32>){ ... }
fn g(q : &Vec<i32>){ ... }
fn main() {
    let mut x = vec![1,2,3];   // x      -+
    g(&x);                     // x       |
    let y = x;                 // y      -+ -+
    f(y);                      // z@f    ----+
    g(&y);                     // none
}
```