Part 3: Linearity

Andrea Pferscher

October 29, 2025

University of Oslo

Recap: Setting up a Type System

- A type syntax (T)
- A subtyping relation ($T <: T'$)
- A typing environment ($\Gamma : \mathsf{Var} \mapsto T$)
- A type judgment ($\Gamma \vdash \mathsf{e} : T$)
- A set of type rules and a notion of type soundness

Topic today: type systems for message-passing concurrency

Recap: Data vs. Behavioral Type, Syntax and Subtyping

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

Goal:

- In channel types, the operations are channel operations
- Specify, document and ensure intended communication patterns
- In the very best case: ensure deadlock freedom

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

Type judgment

To express that statement e is well-typed with type T in environment Γ.

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

Recap: Type Soundness

Type soundness expresses that if the program is well-typed, then evaluation does not block.

- Three intermediate lemmas (error states are not well-typed, preservation, progress)
- Note that we do not ensure termination
- Main consideration for later: Are deadlocked states successfully terminated?

Preservation

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

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

Progress

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

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

Channel Types

Typing Channels

- From now on, we will not fully define a language and give all rules
- Syntax will be Go-like (goroutines, channel operations)
- Real Go-Code will be annotated with `Go`

Mismatched message types

The basic error is that the receiver expects the result to be of a different type than the value the sender sends. The type system of Go can detect such error.

Go

```
c := make(chan int)
go func() { c <- "foo" }()
res := (<-c) + 1
```

```
cannot use "foo" (untyped string constant) as int value in send
```

A Simple Type System for Channels (I/III)

Types

If $\mathtt{T}$ is type then $\mathtt{chan\ T}$ is a type.

Variance

Let $\mathtt{T} <: \mathtt{T}'$, with $\mathtt{T} \neq \mathtt{T}'$. A type constructor $\mathtt{C}$ is

- *Covariant* if $\mathtt{C}(\mathtt{T}) <: \mathtt{C}(\mathtt{T}')$
- *Contravariant* if $\mathtt{C}(\mathtt{T}') <: \mathtt{C}(\mathtt{T})$
- *Invariant* if $\mathtt{C}(\mathtt{T}') \not<: \mathtt{C}(\mathtt{T}) \wedge \mathtt{C}(\mathtt{T}) \not<: \mathtt{C}(\mathtt{T}')$

Subtyping

Channels types are *covariant*: If $\mathtt{T}$ is a subtype of $\mathtt{T}'$ then $\mathtt{chan\ T}$ is a subtype of $\mathtt{chan\ T}'$.

A Simple Type System for Channels (II/III)

Typing send

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

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

Go

```go
type Cat struct{};type Car struct{}
type Animal interface{ name() string }
func main() {
  ch := make(chan Animal)
  go func(c chan Animal) { c <- Cat{} }(ch)}
func (Cat) name() string { return "Meowth" }
```

A Simple Type System for Channels (III/III)

Typing receiving

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

- Essentially the same as calling a method and receiving its result

A Glimpse of Input/Output Modes (I/III)

Beware! The next slides use modified Go-like syntax:

- `<-chan` becomes $\texttt{chan}_?$
- `chan<-` becomes $\texttt{chan}_!$
- `chan` becomes $\texttt{chan}_{!?}$

Modes

- The previous system makes sure the sent data has the right data, but does not consider the direction.
- Modes specify the direction of a channel in a given scope

Go

```go
c := make(chan int)
go func() { c<-1 }()
res := (<-c) + 1
```

A Glimpse of Input/Output Modes (II/III)

Types

Channel types are now annotated with their *mode* or *capability*.

$$T ::= \ldots \mid \texttt{chan}_M\ T \qquad M ::= \,! \mid\, ? \mid\, !?$$

- A channel that can only receive: `?`
- A channel that can only send: `!`
- A channel that allows both: `!?`

Subtyping

We can pass a channel that allows both operation to a more constrained context

$$\texttt{chan}_{!}\ T <: \texttt{chan}_{!?}\ T$$

$$\texttt{chan}_{?}\ T <: \texttt{chan}_{!?}\ T$$

A Glimpse of Input/Output Modes (III/III)

- How to use channels with restricted mode `!?`
- Either use subtyping at every evaluation (like in Go)
- Or use *weakening* to enforce that subtyping relation is used only once
- This ensures that once a channel is used for receiving (sending) once in a thread, then it is only used for receiving (sending) afterwards

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

Weakening

Weakening rule

Allows to make a type less specific. This is *not* just using the T-sub rule – we modify the stored type in the environment.

$$\frac{\Gamma[\mathtt{x} \mapsto \mathtt{T''}] \vdash \mathtt{s} : \mathtt{T} \qquad \mathtt{T''} <: \mathtt{T'}}{\Gamma \cup \{\mathtt{x} \mapsto \mathtt{T'}\} \vdash \mathtt{s} : \mathtt{T}}\ \text{T-weak}$$

Input/Output Modes

Other rules: receive and send with modes

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{chan}_{!}\ \mathtt{T} \qquad \Gamma \vdash \mathtt{v} : \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}}{\Gamma \vdash \mathtt{e}\ \ {<}{-}\ \ \mathtt{v} : \mathtt{Unit}}\ \text{M-send}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{chan}_{?}\mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}}{\Gamma \vdash {<}{-}\ \mathtt{e} : \mathtt{T}}\ \text{M-receive}$$

Important: No subtyping on $\mathtt{chan}_{?}\mathtt{T}'$ and $\mathtt{chan}_{!}\mathtt{T}$. A channel must be weakened before it can be used!

More Channel Types

- Formalizing Γ-splitting and ensuring correct number of uses $\rightarrow$ substructural/**linear types**
- Formalizing order $\rightarrow$ **usage types**
- More expressive protocols and allows different types to be send $\rightarrow$ session types

Learning goals of this lecture:

- How are order and capabilities used to structure concurrency?
- How are order and capabilities described in type systems?
- What parts of type systems must be modified?

Not in this lecture: Full formal treatment and most general cases.

- For this reason the language is a bit simplified.
- No arbitrary expressions, no nested channel types

Linear Types

Linear Types (I/II)

Linearity

The previous systems do not prevent the channels from being used *too little* or *too often*.

```
func main() {
  chn := make(chan!? int)
  <- chn //locks and waits forever
}
```

Linear Types (II/II)

Linearity

The previous systems do not prevent the channels from being used *too little* or *too often*.

```
func main() {
  chn := make(chan!? int)
  go receive(chn)
  chn <- 1
  chn <- 1 //locks and waits forever
}

func receive(c chan? int) int {
  return <-c
}
```

Substructural Linear Types

Linearity

In types, logic and related fields, *linearity* refers to capabilities that are used *exactly once*.

- A linear channel can be used for exactly one send/receive operation
- A linear resource cannot be reused after being accessed, and must be accessed

- Simplifies reasoning about systems because one prohibits reuse in different context.
- In the following: no nested channel operations ($<- <-c$)

Linear Types: Syntax

Type syntax

Let T be a type, and $n, m \in \{0, 1\}$. $\texttt{chan}_{\texttt{?n,!m}}\ \texttt{T}$ is a channel type.
Multiplicity !0 denotes that the channel must not be used for a send operation, !1 that exactly one message must be sent. Analogously for ?.

- $c \mapsto \texttt{chan}_{?1,!1}\ T$ is linear
- $c \mapsto \texttt{chan}_{?0,!0}\ T$ cannot be used anymore
- $c \mapsto \texttt{chan}_{?1,!0}\ T$ receiving possible but no sending anymore
- $c \mapsto \texttt{chan}_{?0,!1}\ T$ sending possible but no receiving anymore

- Subtyping possible, but not needed
- No weakening rule, syntax-driven subtyping

Linear Types: Example

The previous example can be written using linear types, and to forbid multiple accesses.

```
func main() {
  chn := make(chan<?1,!1> int)
  go receive(chn)
  chn <- 1 //chan<?0,!1> int
}

func receive(c chan<?1,!0> int) int {
  return <-c
}
```

Splitting the Environment

```
chn := make(chan<?1,!1> int)
go receive(chn)
```

- Transfer capability to receive messages to new thread
- Limit capabilities for the current thread

- Ensure that no capabilities are remaining
- And catch violation, e.g, `<-c + <-c`

Linear Types: Definition of Splitting Environment

Typing environment

A typing environment Γ can be split into two environments Γ^1, Γ^2 by

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

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

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

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

Linear Types: Definition of 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 an unrestricted environment
- First, sub-system only for expressions

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

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

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

Linear Types for Expressions: Typing Trees Examples

Splitting in arithmetic expressions

We split the environment at every point we descend into subexpressions. Number of splits depends on arity (number of arguments) of operator

$$\frac{\Gamma = \Gamma^1 + \Gamma^2 \qquad \Gamma^1 \vdash e_1 : \texttt{Int} \qquad \Gamma^2 \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}}\ \text{L-add} \qquad \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash -e : \texttt{Int}}\ \text{L-minus}$$

Rules for Booleans operators are analogous

Linear receive

Rule for receiving requires that we are still allowed to receive

$$\frac{\Gamma(\mathtt{v}) = \textbf{chan}_{?1,!0}\ T \qquad \text{un}(\Gamma[\mathtt{v} \mapsto \textbf{chan}_{?0,!0}\ T])}{\Gamma \vdash \leftarrow\!\mathtt{v} : T}\ \text{L-receive}$$

Linear Types for Expressions: Examples

Type safe example:

$$\dfrac{\dfrac{\dfrac{}{\mathrm{un}(\{\mathtt{chan}_{?0,!0}\ \mathtt{int}\})} \quad \dfrac{}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\}(x) = \mathtt{chan}_{?1,!0}\ \mathtt{int}}}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash (<\!-x) : \mathtt{int}} \quad \dfrac{\dfrac{}{\mathrm{un}(\{\mathtt{chan}_{?0,!0}\ \mathtt{int}\})}}{\{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\} \vdash 1 : \mathtt{int}} \quad \dfrac{}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} + \{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\} = \{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\}}}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash (<\!-x) + 1 : \mathtt{int}}$$

No-use prohibited:

$$\dfrac{\dfrac{\mathrm{un}(\{\mathtt{chan}_{?1,!0}\ \mathtt{int}\})}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash 1 : \mathtt{int}} \quad \dfrac{\dfrac{}{\mathrm{un}(\{\mathtt{chan}_{?0,!0}\ \mathtt{int}\})}}{\{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\} \vdash 2 : \mathtt{int}} \quad \dfrac{}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} + \{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\} = \{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\}}}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash 1 + 2 : \mathtt{int}}$$

Double-use prohibited:

$$\dfrac{\dfrac{\dfrac{}{\mathrm{un}(\{\mathtt{chan}_{?0,!0}\ \mathtt{int}\})} \quad \dfrac{}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\}(x) = \mathtt{chan}_{?1,!0}\ \mathtt{int}}}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash (<\!-x) : \mathtt{int}} \quad \dfrac{\dfrac{}{\mathrm{un}(\{\mathtt{chan}_{?0,!0}\ \mathtt{int}\})} \quad \{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\}(x) = \mathtt{chan}_{?1,!0}\ \mathtt{int}}{\{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\} \vdash (<\!-x) : \mathtt{int}} \quad \dfrac{}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} + \{x \mapsto \mathtt{chan}_{?0,!0}\ \mathtt{int}\} = \{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\}}}{\{x \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash (<\!-x) + (<\!-x) : \mathtt{int}}$$

Linear Types for Statements

Termination

- All capabilities must be used up
- Ether before termination (**skip**) or by our last expression (**return**)

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \textbf{skip} : \texttt{Unit}}\ \text{L-skip} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash \mathsf{e} : T \qquad \mathsf{un}(\Gamma_2)}{\Gamma \vdash \textbf{return}\ \mathsf{e} : \texttt{Unit}}\ \text{L-return}$$

Linear Types for Statements: Examples

Example 1:

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

Example 2: Let $\Gamma = \Gamma_1 = \{\mathtt{c} \mapsto \mathbf{chan}_{?1,!0}\ \texttt{Int}\}$, $\Gamma_0 = \{\mathtt{c} \mapsto \mathbf{chan}_{?0,!0}\ \texttt{Int}\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{un}(\Gamma_1[c \mapsto \mathbf{chan}_{?0!0}\ \texttt{Int}])} \qquad \dfrac{}{\Gamma_1(\mathtt{c}) = \mathbf{chan}_{?1,!0}\ \texttt{Int}}}{\Gamma_1 \vdash \mathtt{c} : \mathbf{chan}_{?1,!0}\ \texttt{Int}}}{\Gamma_1 \vdash \,<\!-\mathtt{c} : \texttt{Int}} \qquad \dfrac{}{\mathsf{un}(\Gamma_0)} \qquad \dfrac{}{\Gamma = \Gamma_1 + \Gamma_0}}{\Gamma \vdash \mathbf{return}\ <\!-\mathtt{c} : \texttt{Unit}}$$

Example 3: Let $\Gamma = \Gamma_2 = \{\mathtt{c} \mapsto \mathbf{chan}_{?1,!0}\texttt{Int}, \mathtt{d} \mapsto \mathbf{chan}_{?1,!0}\texttt{Int}\}$, $\Gamma_3 = \{\mathtt{c} \mapsto \mathbf{chan}_{?0,!0}\texttt{Int}, \mathtt{d} \mapsto \mathbf{chan}_{?0,!0}\texttt{Int}\}$

$$\dfrac{\dfrac{\dfrac{\mathsf{un}(\{\mathtt{c} \mapsto \mathbf{chan}_{?0,!0}\texttt{Int}, \mathtt{d} \mapsto \mathbf{chan}_{?1,!0}\texttt{Int}\}) \qquad \dfrac{}{\Gamma_2(\mathtt{c}) = \mathbf{chan}_{?1,!0}\texttt{Int}}}{\Gamma_2 \vdash \mathtt{c} : \mathbf{chan}_{?1,!0}\texttt{Int}}}{\Gamma_2 \vdash \,<\!-\mathtt{c} : \texttt{Int}} \qquad \dfrac{}{\mathsf{un}(\Gamma_3)} \qquad \dfrac{}{\Gamma = \Gamma_2 + \Gamma_3}}{\Gamma \vdash \mathbf{return}\ <\!-\mathtt{c} : \texttt{Unit}}$$

Linear Types for Statements: Sending Rule

Sending (version 1)

- Check that we can send now
- Remove send capability and split the environment into two parts
- One (Γ_1) records the send capability and the capabilities afterwards
- One (Γ_2) record the capabilities of the evaluated expression

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

Linear Types for Statements: Other Rules

- Remaining rules all have the same structure:
- Split environment for each subexpression/substatement
- Propagate split environment into each subexpression/substatement

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{e} : T \qquad \Gamma(\mathtt{v}) = T \qquad \Gamma_2 \vdash \mathtt{s} : \mathtt{Unit}}{\Gamma \vdash \mathtt{v} \ = \ \mathtt{e};\ \mathtt{s} : \mathtt{Unit}}\ \text{L-assign}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 + \Gamma_3 \qquad \Gamma_1 \vdash \mathtt{e} : \mathtt{Bool} \qquad \Gamma_2 \vdash \mathtt{s}_1 : \mathtt{Unit} \qquad \Gamma_2 \vdash \mathtt{s}_2 : \mathtt{Unit} \qquad \Gamma_3 \vdash \mathtt{s}_3 : \mathtt{Unit}}{\Gamma \vdash \mathtt{if(e)\{s_1\}\ else\{s_2\}\ s_3} : \mathtt{Unit}}\ \text{L-branch}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{s}_1 : \mathtt{Unit} \qquad \Gamma_2 \vdash \mathtt{s}_2 : \mathtt{Unit}}{\Gamma \vdash \mathbf{go}\ \mathtt{func()\{s_1\}();\ s_2} : \mathtt{Unit}}\ \text{L-parallel}$$

Example: Linear Types and Sequential Branching

Consider the following environments:

$$\begin{aligned}
\Gamma &= \{\texttt{chn} \mapsto \textbf{chan}_{?1,!1}\ \texttt{Int}\} \\
\Gamma^{?} &= \{\texttt{chn} \mapsto \textbf{chan}_{?1,!0}\ \texttt{Int}\} \\
\Gamma^{!} &= \{\texttt{chn} \mapsto \textbf{chan}_{?0,!1}\ \texttt{Int}\} \\
\Gamma^{0} &= \{\texttt{chn} \mapsto \textbf{chan}_{?0,!0}\ \texttt{Int}\}
\end{aligned}$$

Type-safe:

$$\dfrac{\dfrac{\vdots}{\Gamma^{?} \vdash (\texttt{<-chn}) \geq 0 : \texttt{Bool}} \quad \dfrac{\vdots}{\Gamma^{!} \vdash \texttt{chn <-0} : \texttt{Unit}} \quad \dfrac{\vdots}{\Gamma^{!} \vdash \texttt{chn <-1} : \texttt{Unit}} \quad \dfrac{\vdots}{\Gamma^{0} \vdash \texttt{skip} : \texttt{Unit}} \quad \dfrac{}{\Gamma = \Gamma^{?} + \Gamma^{!} + \Gamma^{0}}}{\Gamma \vdash \texttt{if}((\texttt{<-chn}) \geq 0)\{\texttt{chn <-0}\}\textbf{else}\{\texttt{chn <-1}\}\ \textbf{skip} : \texttt{Unit}}$$

Missed use in branch is detected:

$$\dfrac{\dfrac{\vdots}{\Gamma^{?} \vdash (\texttt{<-chn}) \geq 0 : \texttt{Bool}} \quad \dfrac{\vdots}{\Gamma^{!} \vdash \texttt{chn <-0} : \texttt{Unit}} \quad \dfrac{\vdots}{\Gamma^{!} \vdash \textbf{skip} : \texttt{Unit}} \quad \dfrac{\vdots}{\Gamma^{0} \vdash \texttt{skip} : \texttt{Unit}} \quad \dfrac{}{\Gamma = \Gamma^{?} + \Gamma^{!} + \Gamma^{0}}}{\Gamma \vdash \texttt{if}((\texttt{<-chn}) \geq 0)\{\texttt{chn <-0}\}\textbf{else}\{\textbf{skip}\}\ \textbf{skip} : \texttt{Unit}}$$

Example: Linear Types and Parallelism

We can now, assuming a simple rule for function calls, prove the receiving example.

```
chn := make(chan<?1,!1> int)
go receive(chn)
chn <- 1
```

```
func receive(c chan<?1,!0> int) int {
  return <- c
}
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\quad}\\ \vdots}{\{\mathtt{chn} \mapsto \mathtt{chan}_{?0,!1}\ \mathtt{int}\} \vdash \mathtt{chn} \mathrel{<\!\!-} 1;\ \textbf{skip} : \mathtt{Unit}} \quad \dfrac{\overline{\quad}\\ \vdots}{\{\mathtt{chn} \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash \mathtt{go}\ \mathtt{receive(chn)} : \mathtt{Unit}}}{\{\mathtt{chn} \mapsto \mathtt{chan}_{?0,!1}\ \mathtt{int}\} + \{\mathtt{chn} \mapsto \mathtt{chan}_{?1,!0}\ \mathtt{int}\} \vdash \mathtt{go}\ \mathtt{receive(chn)};\ \mathtt{chn} \mathrel{<\!\!-} 1;\ \textbf{skip} : \mathtt{Unit}}}{\{\mathtt{chn} \mapsto \mathtt{chan}_{?1,!1}\ \mathtt{int}\} \vdash \mathtt{go}\ \mathtt{receive(chn)};\ \mathtt{chn} \mathrel{<\!\!-} 1\ \textbf{skip} : \mathtt{Unit}}}{\{\} \vdash \mathtt{chn} := \mathtt{make(chan} <?1, !1 > \mathtt{int)};\ \mathtt{go}\ \mathtt{receive(chn)};\ \mathtt{chn} \mathrel{<\!\!-} 1;\ \textbf{skip} : \mathtt{Unit}}$$

Type Soundness

Is this enough?

To check that a channel is used exactly once, it is *not* enough to check that the multiplicity is 0 at the end – additionally one must ensure deadlock-freedom!

```
c1 := make(chan<!1,?1> bool)
if(<-c1){ c1 <- true}
```

```
c1 := make(chan<!1,?1> int)
c1 <- (<-c1)
```

Type Soundness: Enforce Parallelism

Sending

- Check that we can send to *but not receive from* c now
- Remove send capability and split the environment into two parts
- One (Γ_1) records the send capability and the capabilities afterwards
- One (Γ_2) records the capabilities of the evaluated expression
- Catches the two single-channel examples from the previous slides

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

Type Soundness

Is this enough?

To check that a channel is used exactly once, it is *not* enough to check that the multiplicity is 0 at the end – additionally one must ensure deadlock-freedom!

```
c1 := make(chan<!1,?1> int)
c2 := make(chan<!1,?1> int)
go func() {v := <−c1; c2 <− 1}()
w := <−c2; c1 <− 1
```

Type Soundness: Assignments

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{e} : T \qquad \Gamma_2' \vdash \mathtt{s} : \mathtt{Unit} \qquad \Gamma(\mathtt{v}) = T}{\Gamma \vdash \mathtt{v} \ = \ \mathtt{e};\ \mathtt{s} : \mathtt{Unit}} \text{ L-assign-DL}$$

- Where Γ_2' sets all receive x in e to $\textbf{chan}_{?0,!0}\ T$ and is Γ_2 otherwise.

$$\forall x.\ \Gamma_1(x) = \textbf{chan}_{?1,!0}\ T \to \Gamma_2'(x) = \textbf{chan}_{?0,!0}$$

- Enforces that when one receives from or sends to a channel, the other capability has been passed to a different thread

Type Soundness

- One can apply the modification of L-assign-DL to all rules
- Guarantee: if system deadlocks more then one channel must be involved.
- Formalized: a state is successfully terminated if (1) all threads are terminated or (2) all threads are stuck or terminated and there are at least 2 stuck threads that waiting on 2 different channels.
- Deadlock analysis can be reduced to relations *between* channels.

```
c1 := make(chan<!1,?1> int)
c2 := make(chan<!1,?1> int)
go func() {v := <-c1; c2 <- 1}()
w := <-c2; c1 <- 1
```

- What else are linear type systems good for?
- Instead of delving into deadlock checkers: can we specify order more elegantly?

Dropping Unrestricted Environments

- What happens if we drop un(Γ) everywhere?

```
c := make(chan<!1,?1> int)
c <- 1
```

- We still have the restriction that we cannot use more then once

Affine types

A variable or channel is *affine* if it is used at most once. A variable or channel is *relevant* if it is used at least once.

- Not very useful for channels
- Useful for other types, e.g., to express that a declared variable may not be used, but if used then only once (for optimizations) or at least once (i.e., no dead declaration)

Other Uses for Linear Types

- Linearity must not be restricted to channel types
- Can be used to detect unused variables (with relevant types)
- Can be modified to be used for resource management
- In particular: every allocation (=declaration) must be paired with a deallocation (=use)

Normal Types and Linear Types in One Language: Syntax

- How to use linear and normal types for channels in one language?
- Idea: use a special symbol to distinguish arbitrary use
- Extend type syntax, environment split and notion of unrestricted environment

Type Syntax

Let T be a type, and $n, m \in \{0, 1, \omega\}$. $\mathtt{chan}_{\mathtt{?n,!m}}\ T$ is a type.
Multiplicity $!\omega$ denotes that the channel can be sent arbitrarily often, analogously for ?.

Normal Types and Linear Types in One Language: Typing Environment

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

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

$$n + n' = n \text{ if } n' = 0$$

$$n + n' = n' \text{ if } n = 0$$

$$n + n' = \omega \text{ otherwise}$$

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

Normal Types and Linear Types in One Language: Adaptions

- Γ is unrestricted if all contained channels have $n = 0$ or $n = \omega$, and $m = 0$ or $m = \omega$.
- A channel is affine if we drop the restriction constraint, but it has been declared with

$$n = m = 1$$

- All rules stay the same except we must exchange every $n = 1$ for $n > 0$ (and same for m)

$$\frac{\Gamma \vdash e : \mathbf{chan}_{?n,!0}\ T \qquad n > 0}{\Gamma \vdash \leftarrow e : T}\ \text{L-receive}$$

Usage Types

Usage Types

- Linear types are not enough to describe protocols
- Consider a channel that is used as a lock
  - Channel is created, token is put it
  - Reading from channel is acquiring token
  - Sending to channel is releasing

Go

```go
func main(){
  global = 0
  lock := make(chan int)
  finish := make(chan int)
  go dual(1, lock, finish)
  go dual(2, lock, finish)
  lock <- 0
  <-finish; <-finish
  <-lock
}
```

Go

```go
func dual(i int, lock chan int,
          finish chan int) {
  <-lock
  //critical here
  lock <- 0
  //non-critical
  <-lock
  //critical here
  finish <- 0; lock <- 0
}
```

Usage Types

What is the type of `lock`? We need something that can express more than linear types!

Go

```
func dual(i int,
      lock chan<?ω,!ω> int,
      finish chan<?0,!1> int) {
  <−lock
  //critical here
  lock <− 0
  //non−critical
  lock <− 0 //bug!
  <−lock
  //critical here
  finish <− 0
  lock <− 0
}
```

Usage Types: Type Syntax

Type syntax

A usage describes the structure of all allowed actions on a channel.

$$
\begin{array}{rll}
T ::= & \ldots\ldots \mid \mathtt{chan}_U\, T & \\
U ::= & 0 & \text{cannot be used} \\
 & \mid\ ?.U & \text{receive} \\
 & \mid\ !.U & \text{send} \\
 & \mid\ U + U & \text{parallel usage} \\
 & \mid\ U \& U & \text{alternative}
\end{array}
$$

- Not considering infinite/arbitrary channel usage ($*U$)

Usage Types: Examples

First receive, then send, then no usage:

$$?.!.0$$

Receive or send, no other usage:

$$?.0\&!.0$$

Use for synchronization once:

$$?.0+!.0$$

Synchronize twice:

$$?.!.0+!.?.0$$

Usage Types: Splitting Type Environment

Splitting environment

Split is *explicit*.

$$\mathtt{chan}_{\mathtt{U}_1+\mathtt{U}_2}\mathtt{T} = \mathtt{chan}_{\mathtt{U}_1}\mathtt{T} + \mathtt{chan}_{\mathtt{U}_2}\mathtt{T}$$

- Also, $0 + 0 = 0$
- The operator $+$ is commutative, so

$$\mathtt{U}_1 + \mathtt{U}_2 = \mathtt{U}_2 + \mathtt{U}_1$$

An environment is unrestricted if all its channels are assigned 0

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash \mathtt{s}_1 : \mathtt{Unit} \qquad \Gamma_2 \vdash \mathtt{s}_2 : \mathtt{Unit}}{\Gamma \vdash \textbf{go func}()\{\ \mathtt{s}_1\}();\ \mathtt{s}_2 : \mathtt{Unit}}\ \text{U-parallel}$$

Splitting Γ: Split Only at Start of New Thread!

Unsound: split at expressions

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \texttt{Int} \qquad \Gamma_2 \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}} \text{ U-add-1}$$

Unsound: propagate

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}} \text{ U-add-2}$$

Sound: match evaluation order on sequence

$$\frac{\Gamma = \Gamma_1 . \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \texttt{Int} \qquad \Gamma_2 \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}} \text{ U-add-3}$$

- Here $\Gamma_1 . \Gamma_2$ is the split along . for all channels used in e_1 and e_2

Usage Types: Rules for Send & Receive

Send

$$\frac{\Gamma + \{c \mapsto \texttt{chan}_{\texttt{U}}\ T\} \vdash \texttt{s}\colon \texttt{Unit} \qquad \Gamma \vdash \texttt{e}\colon T' \qquad T' <: T}{\Gamma + \{c \mapsto \texttt{chan}_{\texttt{!.U}}\ T\} \vdash \texttt{c} \ \texttt{<-e;}\ \texttt{s}\colon \texttt{Unit}}\ \text{U-send}$$

The rule for sending matches on *two* operators

- Sending (<–) is matched on !
- Sequence (;) is matched on .

Receive

This is the rule for receiving from a non-composed expression into a location, which can apply the same matching as for sending.

$$\frac{\Gamma + \{c \mapsto \texttt{chan}_{\texttt{U}}\ T\} \vdash \texttt{s}\colon \texttt{Unit} \qquad \Gamma \vdash \texttt{v}\colon T' \qquad T <: T'}{\Gamma + \{c \mapsto \texttt{chan}_{\texttt{?.U}}\ T\} \vdash \texttt{v =<-c;}\ \texttt{s}\colon \texttt{Unit}}\ \text{U-receive}$$

Splitting Γ: Example

$$
\dfrac{
\dfrac{}{\{c \mapsto \textbf{chan}_{?.0}\ \texttt{Int}\} \vdash (\texttt{<-}\ c) : \texttt{Int}}
\quad
\dfrac{
\dfrac{}{\{c \mapsto \textbf{chan}_{?.0}\ \texttt{Int}\} \vdash (\texttt{<-}\ c) : \texttt{Int}}
\quad
\dfrac{}{\{c \mapsto \textbf{chan}_{0}\ \texttt{Int}\} \vdash 1 : \texttt{Int}}
}{\{c \mapsto \textbf{chan}_{?.0}\ \texttt{Int}\} \vdash (\texttt{<-}\ c + 1) : \texttt{Int}}
\quad
\dfrac{}{\{c \mapsto \textbf{chan}_{?.?.0}\ \texttt{Int}\} = \{c \mapsto \textbf{chan}_{?.0}\ \texttt{Int}\}.\{c \mapsto \textbf{chan}_{?.0}\ \texttt{Int}\}}
}{\{c \mapsto \textbf{chan}_{?.?.0}\ \texttt{Int}\} \vdash (\texttt{<-}\ c) + (\texttt{<-}\ c + 1) : \texttt{Int}}
$$

Usage Types: Running Example (I/IV)

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
  <−lock
}
```

Usage Types: Running Example (II/IV)

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

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

Usage Types: Running Example (III/IV)

- After another split at the two go's

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{\texttt{lock} \mapsto \texttt{chan}_0\ \texttt{int}, \texttt{finish} \mapsto \texttt{chan}_0\ \texttt{int}\} \vdash \texttt{skip} : \mathsf{Unit}}}{\{\texttt{lock} \mapsto \texttt{chan}_{?.0}\ \texttt{int}, \texttt{finish} \mapsto \texttt{chan}_0\ \texttt{int}\} \vdash \texttt{<-lock} : \mathsf{Unit}}}{\{\texttt{lock} \mapsto \texttt{chan}_{?.0}\ \texttt{int}, \texttt{finish} \mapsto \texttt{chan}_{?.0}\ \texttt{int}\} \vdash \texttt{<-finish};\ \texttt{<-lock} : \mathsf{Unit}}}{\{\texttt{lock} \mapsto \texttt{chan}_{?.0}\ \texttt{int}, \texttt{finish} \mapsto \texttt{chan}_{?.?.0}\ \texttt{int}\} \vdash \texttt{<-finish};\ \texttt{<-finish};\ \texttt{<-lock} : \mathsf{Unit}}}{\{\texttt{lock} \mapsto \texttt{chan}_{!.?.0}\ \texttt{int}, \texttt{finish} \mapsto \texttt{chan}_{?.?.0}\ \texttt{int}\} \vdash \texttt{lock <-0};\ \texttt{<-finish};\ \texttt{<-finish};\ \texttt{<-lock} : \mathsf{Unit}}$$

Usage Types: Running Example (IV/IV)

Go

```
func dual(i int,
        lock chan<?.!.?.!.0> int,
        finish chan<!.0> int) {
   <-lock
   //critical here
   lock <- 0
   //non-critical
   lock <- 0 //bug!
   <-lock
   //critical here
   finish <- 0
   lock <- 0
}
```

- Found during typing: receive expected, but send found

$$\{\mathtt{lock} \mapsto \mathtt{chan}_{?.!.0}\ \mathtt{int}, \mathtt{finish} \mapsto \mathtt{chan}_{!.0}\ \mathtt{int}\} \vdash \mathtt{lock} \leftarrow 0; \ldots : \mathtt{Unit}$$

Limitations of Usages

Data types

Cannot express to first send one data type and then another one. e.g., first send a string and then an integer.

Split

Split must be done manually, programmer must ensure that both part match.

$$!.?.0+!.?.0 \qquad ✗$$

In particular with alternative.

$$(!.0\&?.0) + (!.0\&?.0)$$

Wrap-up

This lecture

- Linear types
  - Restrict and control how often operations are performed on value
  - General idea, used beyond channels
- Usage types
  - Explicitly specify order
  - Explicitly specify splits

Next lectures

Concurrency in Rust

Reading: *Type Systems for Concurrent Programs*, Naoki Kobayashi, 2002, Springer LNCS