Part 3: Type Systems and Concurrency

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

October 30, 2023

University of Oslo

Reminder: Setting Up a Type System

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

Topic today: specifics of type systems for message-passing concurrency

Reminder: 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

Big aim:

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

Reminder: 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 $\mathtt{s}$ is well-typed with type T in environment Γ.

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

Reminder: 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

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

Progress

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

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

Reminder: Types for Channels

Typing Writing

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

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

Typing Reading

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

Reminder: 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
}
```

Reminder: Input/Output Modes

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.

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

Other Rules: Read and Write with Modes

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

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

Reminder: We use Go syntax, but all channel types from now on go beyond their type system

More Channel Types

- Formalizing splitting Γ and ensure 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 system?
- 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
- Clear split between statements and expressions

Linear Types

Linear Types

Linearity

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

```
func main() {
  chn := make(chan!? int)
  go read(chn)



}

func read(c chan? int) int {
  return <-c //locks and waits forever
}
```

Linear Types

Linearity

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

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

func read(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

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 written, !1 that it must be written exactly once. Analogously for ?.

- $c : \texttt{chan}_{?1,!1}\ T$ is linear
- $c : \texttt{chan}_{?0,!0}\ T$ cannot be used anymore
- $c : \texttt{chan}_{?1,!0}\ T$ can be read but not written anymore
- $c : \texttt{chan}_{?0,!1}\ T$ can be written but not read anymore

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

Linear Types

Example

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

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

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

Splitting the Environment

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

- Here we must give the capability to read to the new thread
- We must also ensure that our thread does not use this capability anymore

```
chn <- v
  ...
return <-c
```

- Here we must ensure that no use is left over
- And catch corner cases like **return** <−c + <−c

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

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

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

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

Linear Types: Defining Complete Use

Literals and Termination

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

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

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

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

Linear Types: Defining Complete Use

Literals and Termination

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

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

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

$$\frac{\overline{\text{un}(\Gamma)}}{\{c \mapsto \textbf{chan}_{?0,!0}\} \vdash 1 : \texttt{Int}} \qquad \{c \mapsto \textbf{chan}_{?1,!0}\} \vdash 1 : \texttt{Int}$$

Linear Types for Expressions

Splitting in Arithmetic Expressions

We split the environment at every point we descend into subexpressions.

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

- Rules for Booleans are analogous
- Rule for reading requires that we are still allowed to read

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

Linear Types for Expressions

Type safe example

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

No-use prohibited

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

Double-use prohibited

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

Linear Types for Statements

Termination

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

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

Linear Types for Statements

Termination

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

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

$$\frac{\{\mathsf{c} \mapsto \textbf{chan}_{?1,!0}\mathtt{Int}\} \vdash 0 : \mathtt{Unit}}{\{\mathsf{c} \mapsto \textbf{chan}_{?1,!0}\mathtt{Int}\} \vdash \textbf{return}\ 0 : \mathtt{Unit}}$$

Linear Types for Statements

Termination

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

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

Let $\Gamma = \{c \mapsto \textbf{chan}_{?1,!0}\texttt{Int}\}$, $\Gamma_0 = \{c \mapsto \textbf{chan}_{?0,!0}\texttt{Int}\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{un}(\Gamma_0)} \quad \dfrac{}{\Gamma(c) = \textbf{chan}_{?1,!0}\texttt{Int}}}{\Gamma \vdash c : \textbf{chan}_{?1,!0}\texttt{Int}}}{\Gamma \vdash <-c : \texttt{Int}} \quad \dfrac{}{\mathsf{un}(\Gamma_0)} \quad \dfrac{}{\Gamma = \Gamma + \Gamma_0}}{\Gamma \vdash \textbf{return} <-c : \texttt{Unit}}$$

Linear Types for Statements

Termination

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

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

Let $\Gamma = \{\mathsf{c} \mapsto \textbf{chan}_{?0,!1}\mathtt{Int}\}$, $\Gamma_0 = \{\mathsf{c} \mapsto \textbf{chan}_{?0,!0}\mathtt{Int}\}$

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

Linear Types for Statements

Termination

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

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

Let $\Gamma = \{c \mapsto \textbf{chan}_{?1,!0}\texttt{Int}, d \mapsto \textbf{chan}_{?1,!0}\texttt{Int}\}$, $\Gamma_0 = \{c \mapsto \textbf{chan}_{?0,!0}\texttt{Int}, d \mapsto \textbf{chan}_{?0,!0}\texttt{Int}\}$

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

Linear Types for Statements

Writing (unsound, attempt 1)

- Check that we can write 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

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

Linear Types for Statements

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

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

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

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

Example: Linear Types and Sequential Branching

Example

Consider the following environments

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

Type-safe:

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

Missed use in branch is detected:

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

Example: Linear Types and Parallelism

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

```
chn := make(chan<?1,!1> int)
  go { return <-chn}
  chn <- v
  skip
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{\texttt{chn} \mapsto \texttt{chan}_{?0,!1}\ \texttt{int}\} \vdash \texttt{chn} <\!\!-\texttt{v} : \texttt{Unit}} \qquad \dfrac{\vdots}{\{\texttt{chn} \mapsto \texttt{chan}_{?1,!0}\ \texttt{int}\} \vdash \texttt{go read(chn)} : \texttt{Unit}}}{\{\texttt{chn} \mapsto \texttt{chan}_{?0,!1}\ \texttt{int}\} + \{\texttt{chn} \mapsto \texttt{chan}_{?1,!0}\ \texttt{int}\} \vdash \texttt{go read(chn);}\ \texttt{chn} <\!\!-\texttt{v} : \texttt{Unit}}}{\{\texttt{chn} \mapsto \texttt{chan}_{?1,!1}\ \texttt{int}\} \vdash \texttt{go read(chn);}\ \texttt{chn} <\!\!-\texttt{v} : \texttt{Unit}}}{\vdash \texttt{chn} := \texttt{make(chan} <?1, !1> \texttt{int);}\ \texttt{go read(chn);}\ \texttt{chn} <\!\!-\texttt{v} : \texttt{Unit}}$$

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!

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)
c1 <- (<-c1)
```

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

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 – Enforce Parallelism

Writing

- Check that we can write *but now 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

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

Type Soundness – Enforce Parallelism

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

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

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

- Enforces that when one reads or writes from 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 systems 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

- 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 $n, m \in \{0, 1, \omega\}$. $\texttt{chan}_{\texttt{?n,!m}}\ T$ is a type.
Multiplicity $!\omega$ denotes that the channel can be written arbitrarily often. Analogously for ?.

Normal Types and Linear Types in One Language

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

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

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

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

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

- $\texttt{chan}_{?\omega,!\omega} = \texttt{chan}_{?1,!1} + \texttt{chan}_{?\omega,!\omega}$
- $\texttt{chan}_{?\omega,!\omega} = \texttt{chan}_{?0,!0} + \texttt{chan}_{?\omega,!\omega}$
- $\texttt{chan}_{?\omega,!\omega} = \texttt{chan}_{?1,!1} + \texttt{chan}_{?1,!1}$

Normal Types and Linear Types in One Language

- 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)

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

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
  - Writing 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
}
```

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

Go

```
func dual(i int,
       lock chan<?omega,!omega> 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

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

$T ::=$	$\ldots\ldots \mid \texttt{chan}_U T$	
$U ::=$	0	no usage
	$\mid ?.U$	read
	$\mid !.U$	write
	$\mid U + U$	parallel usage
	$\mid U \& U$	alternative

- Inverted view on program: describes behavior from view of a single channel
- Only describes communication over channel, not communication where channel is passed
- Can be extended with repetition (U^*)

Usage Types: Examples

?.!.0

Usage Types: Examples

$$?.!.0$$

First read, then write, then no usage

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

Usage Types: Examples

$$?.!.0$$

First read, then write, then no usage

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

Read or write, no other usage

$$?.0+!.0$$

Usage Types: Examples

$$?.!.0$$

First read, then write, then no usage

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

Read or write, no other usage

$$?.0+!.0$$

Use for synchronization once

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

Usage Types: Examples

$$?.!.0$$

First read, then write, then no usage

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

Read or write, no other usage

$$?.0+!.0$$

Use for synchronization once

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

Synchronize twice.

Usage Types

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

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

Splitting Γ: Split only at start of new thread!

Unsound: Split at expressions

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

Splitting Γ: Split only at start of new thread!

Unsound: Propagate

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

Splitting Γ: Split only at start of new thread!

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

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

Usage Types

Write

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

The rule for writing matches on *two* operators

- Writing ($<\!-$) is matched on !
- Sequence (;) is matched on .

Read

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

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

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

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

Example

- After another split at the two go's

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{\{\mathtt{lock} \mapsto \mathtt{chan}_{0}\ \mathtt{int}, \mathtt{finish} \mapsto \mathtt{chan}_{0}\ \mathtt{int}\} \vdash \mathtt{skip} : \mathtt{Unit}}}{\{\mathtt{lock} \mapsto \mathtt{chan}_{?.0}\ \mathtt{int}, \mathtt{finish} \mapsto \mathtt{chan}_{0}\ \mathtt{int}\} \vdash \mathtt{<-lock} : \mathtt{Unit}}}{\{\mathtt{lock} \mapsto \mathtt{chan}_{?.0}\ \mathtt{int}, \mathtt{finish} \mapsto \mathtt{chan}_{?.0}\ \mathtt{int}\} \vdash \mathtt{<-finish};\ \ \mathtt{<-lock} : \mathtt{Unit}}}{\{\mathtt{lock} \mapsto \mathtt{chan}_{?.0}\ \mathtt{int}, \mathtt{finish} \mapsto \mathtt{chan}_{?.?.0}\ \mathtt{int}\} \vdash \mathtt{<-finish};\ \ \mathtt{<-finish};\ \ \mathtt{<-lock} : \mathtt{Unit}}}{\{\mathtt{lock} \mapsto \mathtt{chan}_{!.?.0}\ \mathtt{int}, \mathtt{finish} \mapsto \mathtt{chan}_{?.?.0}\ \mathtt{int}\} \vdash \mathtt{lock} \mathtt{<-} 0;\ \ \mathtt{<-finish};\ \ \mathtt{<-finish};\ \ \mathtt{<-lock} : \mathtt{Unit}}
$$

Example

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: read expected, but write found

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

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
  - Extension to detect
  - General idea, used beyond channels
- Usage Types
  - Explicitly specify order
  - Explicitly specify splits

Next Lecture

Binary and Multi-Party Session types

Reading: `Type Systems for Concurrent Programs` by Naoki Kobayashi