Part 3: Type Systems and Concurrency

Andrea Pferscher

October 22, 2025

University of Oslo

Types: Foundations

Analyses

Next lectures

- Type systems
- Types for channels
- Ownership and Rust

Reading material

- *Types and Programming Languages*, Benjamin Pierce, 2000, MIT Press
- *Type Systems for Concurrent Programs*, Naoki Kobayashi, 2002, Springer LNCS
- *Uniqueness Typing Simplified*, de Vries et al., 2007, Springer LNCS

Why Types?

- Detecting errors
  - Compiler detects errors before execution (static)
  - Clearer error messages at runtime (dynamic)
  - Enforcing certain programming patterns
- Abstraction
  - Modularity by providing interfaces
  - Hides memory/implementation details
- Documentation/Specification
  - Expresses *intended* behavior
  - Communication with other developers
  - In contrast to comments/documents: enforced to be updated

Why type systems here?

- Insights into compiler construction
- Type systems are a formalization of how developers analyze: How to think about programs?

Foundations of Types

“Well-typed programs cannot go wrong” (Robin Milner, ’78)

- What is a “type”?
- What means “well-typed”?
- What means “go wrong”?
- What kind of type systems exist?
- What does this mean especially for concurrent systems?

Foundations of Types: What is a Type?

"Well-typed programs cannot go wrong"

Types for Expressions

- Types classify expressions
- Expression `e` has a type `T` if `e` will (always) evaluate to a value of type `T`
  - $\{\ldots, -1, 0, 1, \ldots\}$ are values of type `int`
  - $22 + 2$ evaluates to 24, which has type `int`

- Data types of variables are abstractions over memory layout
- What is the type of a function? The type of a channel?
- For us: A type is an abstraction over *data or behavior*
- Channel types are *behavioral types*

Foundations of Types: What is Well-typedness?

"Well-typed programs cannot go wrong"

Type systems

If we know our abstractions, we need to ensure that our program adheres to them.

A type system is a method to check whether a program adheres to its types.

- Dynamic vs. static
  - Static systems check *type annotations* at compile time
  - Dynamic systems check *type tags* at runtime
  - Gradual system check as much as possible statically, and refer the rest to a dynamic system
- Decidable vs. undecidable
  - Static systems should not take too much time, more precise types abstract less
  - Type system can become undecidable (e.g. Java Generics)
- Strong vs. weak typing
  - Strong type systems aim to cover as many possible error sources
  - Weak type systems give more freedom

Foundations of Types: What are Errors? (I/IV)

"Well-typed programs cannot go wrong"

Examples for errors

- General: Applying operators that are not defined on all inputs

```
1+'string' //ill-typed
1+1 //well-typed
...
public Integer f(Integer i) { return 2/i; }
...
f(true) // ill-typed
f(0) // ill-typed?
```

Foundations of Types: What are Errors? (II/IV)

"Well-typed programs cannot go wrong"

Examples for errors

- General: Applying operators that are not defined on all inputs
- Object Orientation (OO): Calling a method that is not supported

```
public class C {
  public Integer f(Integer i) { return i*2; }
}
...
C c = new C();
c.g(1);
```

Foundations of Types: What are Errors? (III/IV)

"Well-typed programs cannot go wrong"

Examples for errors

- General: Applying operators that are not defined on all inputs
- OO: Calling a method that is not supported
- Concurrent: Deadlock

?? How to specify deadlocks? —> channel types

Foundations of Types: What are Errors? (IV/IV)

"Well-typed programs cannot go wrong"

Examples for errors

Not every error is considered a type error. Sometimes the line is not clear, e.g., for null access.

```
public void method(C o){ o.m(); } //Java: Type C allows null
  ...
  this.method(null);
```

```
fun method(o : C){ o.m(); }//Kotlin: Type C does not allow null
  ...
  this.method(null);
```

Type Soundness

"Well-typed programs cannot go wrong"

Type soundness

If a program adheres to its types at compile time, then certain errors do not occur at runtime

- Formalized either as reachability or reduction.
- $e_1 \leadsto e_2$ is one execution/evaluation step from e_1 to e_2

Type soundness as reachability

- A bad operation results in an error state.
- Well-typed programs never reach an error state.

$$(1+1)+1 \leadsto 2+1 \leadsto 1 \qquad \checkmark$$
$$(1+1)+\text{'a'} \leadsto 2+\text{'a'} \leadsto \textbf{error} \qquad \mathcal{X}$$

Type soundness as reduction

- A bad operation blocks the program.
- Well-typed programs never block.

$$(1+1)+1 \leadsto 2+1 \leadsto 1 \qquad \checkmark$$
$$(1+1)+\text{'a'} \leadsto 2+\text{'a'} \qquad \mathcal{X}$$

Type Analysis

How would we analyze this? How would we formally reason about it?

Completeness of Type Systems

Types and logic

Type systems and logics share some properties

- Notions of soundness and completeness
- Judgment (later today)
- Dual use as documentation and specification

Static types

Static type systems are typically incomplete

- In many cases to keep them are decidable
- Their wide adaption hints that the incomplete part is not important in practice

Dynamic types

Dynamic type systems are “complete”, but detect the error too late.

A Simple Type System

A typing discipline consists of

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

Next slides

A simple type system for a simple sequential language.

A Simple Type System

Typing Literal Expressions

Language syntax

Expression e with integer ($n \in \mathbb{Z}$) and Boolean literals:

$$e ::= n \mid \mathit{true} \mid \mathit{false} \mid e + e \mid e \wedge e \mid e \leq e$$

Type syntax

Booleans and integers:

$$T ::= \texttt{Bool} \mid \texttt{Int}$$

- 1
- $1 + 2 \leq 3$
- We allow parentheses if necessary $((1 + 2) \leq 3) \wedge \mathit{true}$

A Simple Type System: Typing Judgment

A judgment is a meta-statement over formal constructs.

Typing judgment

To express that an expression e is well-typed with type T. We write

$$\vdash e : T$$

- Judgment is *true*: $\vdash 1 + 1 : \texttt{Int}$
- Judgment is *false*: $\vdash 1 + 1 : \texttt{Bool}$

A Simple Type System: Typing Rules

Typing rules

- A typing rule contains one conclusion ($\texttt{conclusion}$) and a list of premises ($\texttt{premise}_i$).
- Each conclusion and premise is one judgment
- Its meaning is that if all premises are true, then the conclusion is also true (inference rule)
- A rule without premises is an *axiom* and expresses that something is always true

Notation:

$$\frac{\texttt{premise}_1 \qquad \ldots \qquad \texttt{premise}_n}{\texttt{conclusion}}\ \text{rule name}$$

Our axioms:

$$\frac{}{\vdash \mathit{false} : \texttt{Bool}}\ \text{bool-f} \qquad \frac{}{\vdash \mathit{true} : \texttt{Bool}}\ \text{bool-t} \qquad \frac{}{\vdash n : \texttt{Int}}\ \text{int-literal}$$

A Simple Type System: Expression Rules

The following expresses that if e_1 and e_2 can be typed with boolean type, then so can $e_1 \wedge e_2$.

$$\frac{\vdash e_1 : \texttt{Bool} \qquad \vdash e_2 : \texttt{Bool}}{\vdash e_1 \wedge e_2 : \texttt{Bool}}\ \text{bool-and}$$

The following expresses that if e_1 and e_2 can be typed with integer type, then so can $e_1 + e_2$.

$$\frac{\vdash e_1 : \texttt{Int} \qquad \vdash e_2 : \texttt{Int}}{\vdash e_1 + e_2 : \texttt{Int}}\ \text{int-plus}$$

The following expresses that if e_1 and e_2 can be typed with integer type, then $e_1 \leq e_2$ can be typed with boolean type.

$$\frac{\vdash e_1 : \texttt{Int} \qquad \vdash e_2 : \texttt{Int}}{\vdash e_1 \leq e_2 : \texttt{Bool}}\ \text{bool-leq}$$

A Simple Type System: Typing Tree

A typing rule is a schema that can be applied to a concrete expression. If we do so repeatedly, then the result is a typing tree.

Typing tree

A typing tree is a tree, where each node is a type rule application on a concrete expression.
A tree is closed if all leaves are stemming from axioms.

Rule:

$$\frac{\vdash \mathtt{e}_1 : \mathtt{Int} \qquad \vdash \mathtt{e}_2 : \mathtt{Int}}{\vdash \mathtt{e}_1 + \mathtt{e}_2 : \mathtt{Int}}\ \text{int-plus}$$

Rule application:

$$\frac{\vdash 12 : \mathtt{Int} \qquad \vdash 13 : \mathtt{Int}}{\vdash 12 + 13 : \mathtt{Int}}\ \text{int-plus}$$

A Simple Type System: Example

$$\cfrac{\cfrac{\cfrac{}{\vdash 1 : \texttt{Int}}\,\text{int-literal} \quad \cfrac{}{\vdash 2 : \texttt{Int}}\,\text{int-literal}}{\vdash 1 + 2 : \texttt{Int}}\,\text{int-plus} \qquad \cfrac{}{\vdash 3 : \texttt{Int}}\,\text{int-literal}}{\vdash (1+2) \leq 3 : \texttt{Bool}}\,\text{bool-leq}$$

This means that $1 + 2 \leq 3$ indeed has type $\texttt{Bool}$.

$$\cfrac{\cfrac{\vdash \textit{true} : \texttt{Int} \quad \cfrac{}{\vdash 2 : \texttt{Int}}\,\text{int-literal}}{\vdash \textit{true} + 2 : \texttt{Int}}\,\text{int-plus} \qquad \cfrac{}{\vdash 3 : \texttt{Int}}\,\text{int-literal}}{\vdash \textit{true} + 2 \leq 3 : \texttt{Bool}}\,\text{bool-leq}$$

This means that $\textit{true} + 2 \leq 3$ does not have type $\texttt{Bool}$.

A Simple Type System: Termination

We have types and typing rules, for type soundness we also need expression evaluation.

Evaluation

We do not define evaluation formally here, but assume that $e_1 \rightsquigarrow e_2$ is one execution/evaluation step from e_1 to e_2.

- $1 + 2 \rightsquigarrow 3$
- $1 + 2 \leq 5 \rightsquigarrow 3 \leq 5$
- $3 \leq 3 \rightsquigarrow \mathit{true}$

Literals and termination

An evaluation of expression e_1 *successfully terminates*, if

$$e_1 \rightsquigarrow \cdots \rightsquigarrow e_{\text{final}}$$

and e_{final} is either a literal, e.g. an integer literal n or a boolean literal (true, false), or if e_1 is one of these expressions itself.

A Simple Type System: Type Soundness

Type soundness

Typically, *soundness* (also called *safety*) requires three properties:

- All expressions that are successfully terminated are well-typed
- If a well-typed expression can evaluate, then the result is well-typed (preservation/reduction)
- If a well-typed expression is not successfully terminated, then it can evaluate (progress)

Together, these properties imply that if an expression is well-typed, and its evaluation terminates, then it terminates successfully.

A Simple Type System: Preservation & Progress

Preservation

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

$$\forall \mathsf{e}, \mathsf{e}', \mathtt{T}.\ \big((\mathsf{e} : \mathtt{T} \wedge \mathsf{e} \rightsquigarrow \mathsf{e}') \rightarrow \mathsf{e}' : \mathtt{T}\big)$$

Progress

If a well-typed expression is not successfully terminated (term(e)), then it can evaluate

$$\forall \mathsf{e}, \mathtt{T}.\ \big((\mathsf{e} : \mathtt{T} \wedge \neg \mathsf{term}(\mathsf{e})) \rightarrow \exists \mathsf{e}'.\ \mathsf{e} \rightsquigarrow \mathsf{e}'\big)$$

- Preservation states that typeability is an invariant
- Progress is almost deadlock freedom, typically harder to proof
- More general formulations possible

Typing Environment and Subtyping

A Simple Type Environment

- We can now type simple expressions
- How do we move towards types for concurrency?
- Next two ingredients:
- Typing of variables
  - Typing variables requires to keep track of which variables are declared
  - We will record information in a *type environment*
- Subtyping
  - We will introduce a second judgment to express the relation between types
- Typing environments and subtyping relations are critical for channel types

A Simple Type Environment: Variables

Language syntax

Expressions with integer and boolean literals:

$$e ::= n \mid \mathit{true} \mid \mathit{false} \mid e + e \mid e \wedge e \mid e \leq e \mid v$$

Type syntax (unchanged)

Booleans and integers:

$$T ::= \texttt{Bool} \mid \texttt{Int}$$

- $\texttt{v}$
- $1 + \texttt{v} \leq 3$
- We allow parentheses if necessary $((1 + \texttt{v}) \leq 3) \wedge \texttt{w}$

A Simple Type Environment: Definition

Type environment

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

- Notation to access the type of a variable $\mathtt{v}$ in environment Γ: $\Gamma(\mathtt{v})$
- Notation for an environment with two integer variables $\mathtt{v}, \mathtt{w}$:

$$\Gamma = \{\mathtt{v} \mapsto \mathtt{Int}, \mathtt{w} \mapsto \mathtt{Int}\}$$

  An empty type environment is denoted $\Gamma = \emptyset$.
- Notation for updating the environment

$$\Gamma[\mathtt{x} \mapsto \mathtt{T}] = \Gamma'$$

  , where $\Gamma'(\mathtt{x}) = \mathtt{T}$ and $\Gamma'(\mathtt{y}) = \Gamma(\mathtt{y})$ for all other variables $\mathtt{y} \neq \mathtt{x}$.
- Notation if a variable has no assigned type

$$\Gamma(\mathtt{x}) = \bot$$

A Simple Type Environment: Type Judgment

Type judgment

The type judgment includes the type environment:

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

This reads as *expression* $\mathtt{e}$ *has type* $\mathtt{T}$ *if all variables are as described by* Γ.

New rule: The premise is a new judgment that holds iff the equality holds.

$$\frac{\Gamma(\mathtt{v}) = \mathtt{T}}{\Gamma \vdash \mathtt{v} : \mathtt{T}}\ \text{var}$$

The type environment is added to all other rules and carried over from conclusion to premises. For example:

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{Bool}}{\Gamma \vdash \mathtt{e}_1 \land \mathtt{e}_2 : \mathtt{Bool}}\ \text{bool-and}$$

A Simple Type Environment: Examples

Typing now depends on the type of the variables. Let $\Gamma_1 = \{\mathtt{v} \mapsto \mathtt{Int}\}, \Gamma_2 = \emptyset$

$$\dfrac{\dfrac{}{\Gamma_1 \vdash 1 : \mathtt{Int}}\ \text{int-literal} \qquad \dfrac{\dfrac{}{\Gamma_1(\mathtt{v}) = \mathtt{Int}}}{\Gamma_1 \vdash \mathtt{v} : \mathtt{Int}}\ \text{var}}{\dfrac{\Gamma_1 \vdash 1 + \mathtt{v} : \mathtt{Int}}{}}\ \text{int-plus} \qquad \dfrac{}{\Gamma_1 \vdash 3 : \mathtt{Int}}\ \text{int-literal}$$

$$\dfrac{\Gamma_1 \vdash 1 + \mathtt{v} : \mathtt{Int} \qquad \Gamma_1 \vdash 3 : \mathtt{Int}}{\Gamma_1 \vdash (1 + \mathtt{v}) \leq 3 : \mathtt{Bool}}\ \text{bool-leq}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash 1 : \mathtt{Int}}\ \text{int-literal} \qquad \dfrac{\Gamma_2(\mathtt{v}) = \mathtt{Int}}{\Gamma_2 \vdash \mathtt{v} : \mathtt{Int}}\ \text{var}}{\Gamma_2 \vdash 1 + \mathtt{v} : \mathtt{Int}}\ \text{int-plus} \qquad \dfrac{}{\Gamma_2 \vdash 3 : \mathtt{Int}}\ \text{int-literal}}{\Gamma_2 \vdash 1 + \mathtt{v} \leq 3 : \mathtt{Bool}}\ \text{bool-leq}$$

A Simple Type Environment: Evaluation

Evaluation

Let σ be map from variables to literals. We do not define evaluation formally here, but assume that $\mathtt{e}_1 \leadsto_\sigma \mathtt{e}_2$ is one execution/evaluation step from $\mathtt{e}_1$ to $\mathtt{e}_2$. In particular, $\mathtt{v} \leadsto_\sigma \sigma(\mathtt{v})$.

A Simple Type Environment: Preservation & Progress

Preservation

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

$$\forall \Gamma, \mathrm{e}, \mathrm{e}', \mathtt{T}.\ \big((\Gamma \vdash \mathrm{e} : \mathtt{T} \wedge \mathrm{e} \rightsquigarrow \mathrm{e}') \rightarrow \Gamma \vdash \mathrm{e}' : \mathtt{T}\big)$$

Progress

If a well-typed expression is not successfully terminated (term(e)), then it can evaluate

$$\forall \Gamma, \mathrm{e}, \mathtt{T}.\ \big((\Gamma \vdash \mathrm{e} : \mathtt{T} \wedge \neg \mathrm{term}(\mathrm{e})) \rightarrow \exists \mathrm{e}'.\ \mathrm{e} \rightsquigarrow \mathrm{e}'\big)$$

- Additionally, we must ensure that σ, adheres to Γ
- For every variable $\mathtt{v}$ we must have $\Gamma \vdash \sigma(\mathtt{v}) : \Gamma(\mathtt{v})$

Simple Subtyping

- Let us introduce a simple subtype of rational numbers: integers
- We need to extend the type syntax, adjust the typing rules and formalize subtyping
- Subtyping is formalized as a special type

Type syntax

Booleans, integers and rational numbers:

$$T ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Number}$$

$$\frac{n \in \mathbb{Z}}{\vdash n : \texttt{Int}} \text{ int-literal}$$

Simple Subtyping: Rules

We introduce a new judgment to express that $\texttt{T}_1$ is a subtype of $\texttt{T}_2$: $\texttt{T}_1 <: \texttt{T}_2$

Reflexivity and transitivity

Every type is a subtype of itself (reflexive), subtyping is transitive

$$\frac{}{\texttt{T} <: \texttt{T}}\ \text{T-refl} \qquad \frac{\texttt{T}_1 <: \texttt{S} \qquad \texttt{S} <: \texttt{T}_2}{\texttt{T}_1 <: \texttt{T}_2}\ \text{T-trans}$$

Core rules

The actual subtyping rules are specific for the language, for us it is just this one

$$\frac{}{\texttt{Int} <: \texttt{Number}}\ \text{T-int}$$

Application

At every point during type-checking, we can chose to use a subtype

$$\frac{\texttt{S} <: \texttt{T} \qquad \Gamma \vdash \texttt{e} : \texttt{S}}{\Gamma \vdash \texttt{e} : \texttt{T}}\ \text{T-sub}$$

Simple Subtyping: Example

Now we can type the literal 1 with `Int` using the new rules

$$\dfrac{\dfrac{}{\texttt{Int} <: \texttt{Number}}\ \text{T-int} \qquad \dfrac{\dfrac{}{1 \in \mathbb{Z}}}{\emptyset \vdash 1 : \texttt{Int}}\ \text{int-literal}}{\emptyset \vdash 1 : \texttt{Number}}\ \text{T-sub}$$

- Soundness is not affected by subtyping
- Rule T-sub is not *syntax-directed*
  - Can always be applied
  - Requires to chose a suitable `S`
  - Hard to implement in an algorithm
- This is orthogonal to concurrency, Pierce (Ch. 16) has details on algorithmic subtyping

Syntax-directed Subtyping

- Instead of T-sub, we can allow subtyping in other rules
- In the rest of the lecture, we do no use T-sub

$$\frac{\vdash \mathtt{e}_1 : T_1 \qquad T_1 <: \mathtt{Number} \qquad \vdash \mathtt{e}_2 : T_2 \qquad T_2 <: \mathtt{Number}}{\vdash \mathtt{e}_1 + \mathtt{e}_2 : \mathtt{Number}} \text{ number-plus}$$

Types for Statements

Syntax

Language

Expressions are as before, statements are a simple imperative language

$$\mathtt{s} ::= \mathtt{v} \; = \; \mathtt{e;s} \mid \mathtt{T\ v} \; = \; \mathtt{e;\ s}$$
$$\mid \textbf{skip} \mid \textbf{if}(\mathtt{e})\{\mathtt{s}\}\mathtt{s}$$

Type syntax

Integers, rational number, booleans, unit type. Subtyping as before.

$$T ::= \mathtt{Int} \mid \mathtt{Number} \mid \mathtt{Bool} \mid \mathtt{Unit}$$

- Unit type is used to type statements
- A statement has unit type if it is typeable, and no type if it is not typeable
- Akin to **void** in Java
- No subtype relation to any other type

Type System (I/II)

Rules for expressions are as before.

Simple Statements

Skip is always well-typed

$$\frac{}{\Gamma \vdash \textbf{skip} : \mathtt{Unit}}\ \text{skip}$$

Assignment

Assignment checks that the type of the expression is a subtype of the variable, and that the continuation is typeable. Note that this also checks that the variable is declared – otherwise $\Gamma(\mathtt{v}) = \perp$ and the second premise fails.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \mathtt{S} <: \Gamma(\mathtt{v}) \qquad \Gamma \vdash \mathtt{s} : \mathtt{Unit}}{\Gamma \vdash \mathtt{v}\ =\ \mathtt{e};\ \mathtt{s} : \mathtt{Unit}}\ \text{assign}$$

Type System (II/II)

Declaration

Declaration is as before, but additionally updates the environment for the continuation.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T} \qquad \Gamma[\mathtt{v} \mapsto \mathtt{T}] \vdash \mathtt{s} : \mathtt{Unit}}{\Gamma \vdash \mathtt{T}\ \mathtt{v}\ =\ \mathtt{e};\ \mathtt{s} : \mathtt{Unit}}\ \text{decl}$$

Branching

Branching checks that the condition has boolean type, and both conditional statement and continuation. This implements scoping: if the environment get updated by $\mathtt{s}_1$, then these declarations are lost for $\mathtt{s}_2$.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{s}_1 : \mathtt{Unit} \qquad \Gamma \vdash \mathtt{s}_2 : \mathtt{Unit}}{\Gamma \vdash \textbf{if}(\mathtt{e})\{\mathtt{s}_1\}\mathtt{s}_2 : \mathtt{Unit}}\ \text{branch}$$

Soundness (I/II)

- Important: terminated program must be well-typed!
- If one uses the error state, it *must not* be well-typed.

Type soundness

If statement `s` can be typed with `Unit`, and its execution terminates, then it terminates with `s` is fully reduced to **skip**.

$$
\begin{aligned}
&\texttt{Int}\ v = 1; v = v + 2 & \\
\rightsquigarrow\ & v = v + 2 & \sigma = \{v \mapsto 1\} \\
\rightsquigarrow\ & \textbf{skip} & \sigma = \{v \mapsto 3\}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Int}\ v = 1; v = v + \mathit{true} & \\
\rightsquigarrow\ & v = v + \mathit{true} & \sigma = \{v \mapsto 1\}
\end{aligned}
$$

Soundness (II/II)

Remarks

- Usual preservation and progress properties
- Initial typing starts with empty environment, i.e., no declared variables
- Each branch and programs ends in skip.

Environment Example

Environment

A variable $\mathtt{v}$ must be declared to be recorded in the environment, otherwise any rule that tries to evaluate $\Gamma(\mathtt{v})$ fails.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdots}}{\{\mathtt{v} \mapsto \mathtt{Int}\} \vdash \mathtt{v} + 2 : \mathtt{Int}} \quad \dfrac{}{\mathtt{Int} <: \mathtt{Int}} \quad \dfrac{}{\{\mathtt{v} \mapsto \mathtt{Int}\} \vdash \textbf{skip} : \mathtt{Unit}}}{\{\mathtt{v} \mapsto \mathtt{Int}\} \vdash \mathtt{v} = \mathtt{v} + 2; \textbf{skip} : \mathtt{Unit}} \quad \dfrac{\dfrac{}{1 \in \mathbb{Z}}}{\emptyset \vdash 1 : \mathtt{Int}} \quad \dfrac{}{\mathtt{Int} <: \mathtt{Int}}}{\emptyset \vdash \mathtt{Int}\ \mathtt{v} = 1; \mathtt{v} = \mathtt{v} + 2; \textbf{skip} : \mathtt{Unit}}$$

$$\dfrac{\dfrac{\{\mathtt{v} \mapsto \mathtt{Int}\} \vdash \mathtt{w} : \mathtt{Int} \quad \dfrac{}{\mathtt{Int} <: \mathtt{Int}} \quad \dfrac{}{\{\mathtt{v} \mapsto \mathtt{Int}\} \vdash \textbf{skip} : \mathtt{Unit}}}{\{\mathtt{v} \mapsto \mathtt{Int}\} \vdash \mathtt{v} = \mathtt{w}; \textbf{skip} : \mathtt{Unit}} \quad \dfrac{\dfrac{}{1 \in \mathbb{Z}}}{\emptyset \vdash 1 : \mathtt{Int}} \quad \dfrac{}{\mathtt{Int} <: \mathtt{Int}}}{\emptyset \vdash \mathtt{Int}\ \mathtt{v} = 1; \mathtt{v} = \mathtt{w}; \textbf{skip} : \mathtt{Unit}}$$

Example

Scoping

Scoping is implemented by not transferring the updated environment. In our rule for branching, we type the continuation with the type environment *before* the branching – all variables declared within are lost.

$$\dfrac{\dfrac{}{\emptyset \vdash \mathit{true} : \mathtt{Bool}} \quad \dfrac{\dfrac{}{\vdots}}{\emptyset \vdash \mathtt{Int\ v} = 1;\ \textbf{skip} : \mathtt{Unit}} \quad \emptyset \vdash \mathtt{v} = 2; \textbf{skip} : \mathtt{Unit}}{\emptyset \vdash \textbf{if}(\mathit{true})\{\mathtt{Int\ v} = 1;\ \textbf{skip}\}\mathtt{v} = 2; \textbf{skip} : \mathtt{Unit}}$$

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. Implemented in Go.

Go

```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: Variance

Types

If $\mathtt{T}$ is type then $\mathtt{chan}\ \mathtt{T}$ 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 $\mathtt{chan}\ \mathtt{T}$ is a subtype of $\mathtt{chan}\ \mathtt{T}'$.

A Simple Type System for Channels: Rule for Writing

Typing writing

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

- 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: Rule for Reading

Typing reading

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

- Essentially the same as calling a method and reading 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 read/receive: `?`
- A channel that can only write/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 reading (writing) once in a thread, then it is only used for reading (writing) afterwards

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

Input/Output Modes: Rules (I/II)

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: Receive and send with modes

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

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

Input/Ouput Modes: Rules (II/II)

Other rules: receive and send with modes

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

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

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

Next Lecture: Linear types

Wrap-up

Today's lecture

- General structure of static type systems
- Simple type systems for channels
- Introduction: modes

Upcoming lectures

- More on modes
- More complex channel types
    - Linear types
- Uniqueness types, towards the ownership system of Rust