



Semantics and Verification of Software

Winter Semester 2017/18

Lecture 18: Fairness in CSP & Type Systems

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<http://moves.rwth-aachen.de/teaching/ws-1718/sv-sw/>

Recap: Channel Communication

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

Recap: Channel Communication

Syntax of CSP

Definition (Syntax of CSP)

The syntax of CSP is given by

$$\begin{aligned} a &::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp \\ b &::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp \\ c &::= \text{skip} \mid x := a \mid \alpha ? x \mid \alpha ! a \mid \\ &\quad c_1 ; c_2 \mid \text{if } gc \text{ fi} \mid \text{do } gc \text{ od} \mid c_1 \parallel c_2 \in Cmd \\ gc &::= b \rightarrow c \mid b \wedge \alpha ? x \rightarrow c \mid b \wedge \alpha ! a \rightarrow c \mid gc_1 \square gc_2 \in GCmd \end{aligned}$$

- In $c_1 \parallel c_2$, commands c_1 and c_2 must **not use common variables** (only local store)
- **Guarded command** $gc_1 \square gc_2$ represents an **alternative**
- In $b \rightarrow c$, b acts as a **guard** that enables the execution of c only if evaluated to **true**
- $b \wedge \alpha ? x \rightarrow c$ and $b \wedge \alpha ! a \rightarrow c$ additionally require the respective I/O operation to be enabled
- If none of its alternatives is enabled, a guarded command gc **fails** (configuration **fail**)
- **if** nondeterministically picks an enabled alternative
- A **do** loop is iterated until its body fails

Recap: Channel Communication

Semantics of CSP I

Definition (Semantics of CSP – Commands (*Cmd*))

$$\begin{array}{c} \frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \langle \downarrow, \sigma \rangle} \\ \frac{}{\langle \alpha?x, \sigma \rangle \xrightarrow{\alpha?z} \langle \downarrow, \sigma[x \mapsto z] \rangle} \\ \frac{\langle c_1, \sigma \rangle \xrightarrow{\lambda} \langle c'_1, \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \xrightarrow{\lambda} \langle c'_1; c_2, \sigma' \rangle} \\ \frac{\langle gc, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle \text{do } gc \text{ od}, \sigma \rangle \xrightarrow{\lambda} \langle c; \text{do } gc \text{ od}, \sigma' \rangle} \\ \frac{\langle c_1, \sigma \rangle \xrightarrow{\lambda} \langle c'_1, \sigma' \rangle}{\langle c_1 \parallel c_2, \sigma \rangle \xrightarrow{\lambda} \langle c'_1 \parallel c_2, \sigma' \rangle} \\ \frac{\langle c_1, \sigma \rangle \xrightarrow{\alpha?z} \langle c'_1, \sigma' \rangle \quad \langle c_2, \sigma \rangle \xrightarrow{\alpha!z} \langle c'_2, \sigma' \rangle}{\langle c_1 \parallel c_2, \sigma \rangle \rightarrow \langle c'_1 \parallel c'_2, \sigma' \rangle} \end{array} \quad \begin{array}{c} \frac{\langle a, \sigma \rangle \rightarrow z}{\langle x := a, \sigma \rangle \rightarrow \langle \downarrow, \sigma[x \mapsto z] \rangle} \\ \frac{\langle a, \sigma \rangle \rightarrow z}{\langle \alpha!a, \sigma \rangle \xrightarrow{\alpha!z} \langle \downarrow, \sigma \rangle} \\ \frac{\langle gc, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle \text{if } gc \text{ fi}, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle} \\ \frac{\langle gc, \sigma \rangle \rightarrow \text{fail}}{\langle \text{do } gc \text{ od}, \sigma \rangle \rightarrow \langle \downarrow, \sigma \rangle} \\ \frac{\langle c_2, \sigma \rangle \xrightarrow{\lambda} \langle c'_2, \sigma' \rangle}{\langle c_1 \parallel c_2, \sigma \rangle \xrightarrow{\lambda} \langle c_1 \parallel c'_2, \sigma' \rangle} \\ \frac{\langle c_1, \sigma \rangle \xrightarrow{\alpha!z} \langle c'_1, \sigma' \rangle \quad \langle c_2, \sigma \rangle \xrightarrow{\alpha?z} \langle c'_2, \sigma' \rangle}{\langle c_1 \parallel c_2, \sigma \rangle \rightarrow \langle c'_1 \parallel c'_2, \sigma' \rangle} \end{array}$$

Recap: Channel Communication

Semantics of CSP II

Definition (Semantics of CSP – Guarded commands (*GCmd*))

$$\begin{array}{c} \frac{\langle b, \sigma \rangle \rightarrow \text{true}}{\langle b \rightarrow c, \sigma \rangle \rightarrow \langle c, \sigma \rangle} \qquad \frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle b \rightarrow c, \sigma \rangle \rightarrow \text{fail}} \\ \frac{\langle b, \sigma \rangle \rightarrow \text{true}}{\langle b \wedge \alpha?x \rightarrow c, \sigma \rangle \xrightarrow{\alpha?z} \langle c, \sigma[x \mapsto z] \rangle} \qquad \frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle b \wedge \alpha?x \rightarrow c, \sigma \rangle \rightarrow \text{fail}} \\ \frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle a, \sigma \rangle \rightarrow z}{\langle b \wedge \alpha!a \rightarrow c, \sigma \rangle \xrightarrow{\alpha!z} \langle c, \sigma \rangle} \qquad \frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle b \wedge \alpha!a \rightarrow c, \sigma \rangle \rightarrow \text{fail}} \\ \frac{\langle gc_1, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle gc_1 \square gc_2, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle} \qquad \frac{\langle gc_2, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle gc_1 \square gc_2, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle} \\ \frac{\langle gc_1, \sigma \rangle \rightarrow \text{fail} \quad \langle gc_2, \sigma \rangle \rightarrow \text{fail}}{\langle gc_1 \square gc_2, \sigma \rangle \rightarrow \text{fail}} \end{array}$$

Fairness in CSP

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

Fairness I

- Informally: **unfair** behaviour excludes processes from being executed
- Here: consider parallel composition of $n \geq 1$ sequential programs with executions of the form $\kappa_0 \rightarrow \kappa_1 \rightarrow \kappa_2 \rightarrow \dots$ where $\kappa_j = \langle c_1^{(j)} \parallel \dots \parallel c_n^{(j)}, \sigma_j \rangle$ and, for some $1 \leq i \leq n$ and $k_0 \in \mathbb{N}$, $c_i^{(k)} = c_i^{(k_0)}$ for all $k \geq k_0$
- But: only unfair if c_i enabled

Fairness I

- Informally: **unfair** behaviour excludes processes from being executed
- Here: consider parallel composition of $n \geq 1$ sequential programs with executions of the form $\kappa_0 \rightarrow \kappa_1 \rightarrow \kappa_2 \rightarrow \dots$ where $\kappa_j = \langle c_1^{(j)} \parallel \dots \parallel c_n^{(j)}, \sigma_j \rangle$ and, for some $1 \leq i \leq n$ and $k_0 \in \mathbb{N}$, $c_i^{(k)} = c_i^{(k_0)}$ for all $k \geq k_0$
- But: only unfair if c_i enabled

Definition 18.1 (Enabledness)

c_i is **enabled** in configuration $\kappa = \langle c_1 \parallel \dots \parallel c_n, \sigma \rangle$ if there exists $\kappa' = \langle c'_1 \parallel \dots \parallel c'_n, \sigma' \rangle$ with $\kappa \rightarrow \kappa'$ and $c'_i \neq c_i$.

Fairness I

- Informally: **unfair** behaviour excludes processes from being executed
- Here: consider parallel composition of $n \geq 1$ sequential programs with executions of the form $\kappa_0 \rightarrow \kappa_1 \rightarrow \kappa_2 \rightarrow \dots$ where $\kappa_j = \langle c_1^{(j)} \parallel \dots \parallel c_n^{(j)}, \sigma_j \rangle$ and, for some $1 \leq i \leq n$ and $k_0 \in \mathbb{N}$, $c_i^{(k)} = c_i^{(k_0)}$ for all $k \geq k_0$
- But: only unfair if c_i enabled

Definition 18.1 (Enabledness)

c_i is **enabled** in configuration $\kappa = \langle c_1 \parallel \dots \parallel c_n, \sigma \rangle$ if there exists $\kappa' = \langle c'_1 \parallel \dots \parallel c'_n, \sigma' \rangle$ with $\kappa \rightarrow \kappa'$ and $c'_i \neq c_i$.

Example 18.2

1. $x := 0$ enabled in $\langle x := 0 \parallel y := 1, \sigma \rangle$ (actually always enabled)
2. $\alpha?x$ enabled in $\langle \alpha?x \parallel \alpha!0, \sigma \rangle$
3. $\alpha?x$ not enabled in $\langle \alpha?x \parallel \beta!1, \sigma \rangle$

Fairness II

Definition 18.3 (Fairness)

An execution $\kappa_0 \rightarrow \kappa_1 \rightarrow \kappa_2 \rightarrow \dots$ where $\kappa_j = \langle c_1^{(j)} \parallel \dots \parallel c_n^{(j)}, \sigma_j \rangle$ and, for some $1 \leq i \leq n$ and $k_0 \in \mathbb{N}$, $c_i^{(k)} = c_i^{(k_0)}$ for all $k \geq k_0$ is called

1. **strongly unfair** if $c_i^{(k)}$ is enabled in κ_k for all $k \geq k_0$
2. **weakly unfair** if $c_i^{(k)}$ is enabled in κ_k for infinitely many $k \geq k_0$

Fairness III

Example 18.4

1. $\langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle$
 $\rightarrow \langle x := x + 1; \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle$
 $\rightarrow \langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle \rightarrow \dots$

is strongly unfair since $y := y + 1$ is always enabled

Fairness III

Example 18.4

1. $\langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle$
 $\rightarrow \langle x := x + 1; \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle$
 $\rightarrow \langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle \rightarrow \dots$

is strongly unfair since $y := y + 1$ is always enabled

2. $\langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel \alpha!1 \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle x := x + 1; \text{do true} \rightarrow x := x + 1 \text{ od} \parallel \alpha!1 \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel \alpha!1 \parallel \alpha?y, \dots \rangle \rightarrow \dots$

is strongly unfair since both I/O operations are always enabled

Fairness III

Example 18.4

1. $\langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle$
 $\rightarrow \langle x := x + 1; \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle$
 $\rightarrow \langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel y := y + 1, \dots \rangle \rightarrow \dots$

is strongly unfair since $y := y + 1$ is always enabled

2. $\langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel \alpha!1 \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle x := x + 1; \text{do true} \rightarrow x := x + 1 \text{ od} \parallel \alpha!1 \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle \text{do true} \rightarrow x := x + 1 \text{ od} \parallel \alpha!1 \parallel \alpha?y, \dots \rangle \rightarrow \dots$

is strongly unfair since both I/O operations are always enabled

3. $\langle \text{do } \alpha!1 \rightarrow \text{skip} \text{ od} \parallel \text{do } \alpha?x \rightarrow \text{skip} \text{ od} \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle \text{skip}; \text{do } \alpha!1 \rightarrow \text{skip} \text{ od} \parallel \text{skip}; \text{do } \alpha?x \rightarrow \text{skip} \text{ od} \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle \text{skip}; \text{do } \alpha!1 \rightarrow \text{skip} \text{ od} \parallel \text{do } \alpha?x \rightarrow \text{skip} \text{ od} \parallel \alpha?y, \dots \rangle$
 $\rightarrow \langle \text{do } \alpha!1 \rightarrow \text{skip} \text{ od} \parallel \text{do } \alpha?x \rightarrow \text{skip} \text{ od} \parallel \alpha?y, \dots \rangle \rightarrow \dots$

is weakly unfair since $\alpha?y$ is (only) enabled in every third configuration

Summary: Nondeterminism and Parallelism

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

Summary: Nondeterminism and Parallelism

Summary: Nondeterminism and Parallelism

- Important modelling aspects:
 - **parallelism** (here: interleaving = nondeterminism + sequential execution)
 - **interaction** (here: via shared variables/channels)
- Interleaving requires **small-step execution relation**
- **Communication** between parallel processes is represented by labels on transitions
- Parallelism raises new issues such as **fairness**

Types

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

Types

Why Types?

The goal

Type systems aim to avoid software design mistakes.

Types

Why Types?

The goal

Type systems aim to avoid software design mistakes.

Kinds of type systems

The good: Static types that guarantee absence of (certain) runtime faults (example: no memory access errors in Java)

The bad: Static types that have mostly decorative value but do not provide runtime guarantees (example: C, C++)

The ugly: Dynamic types that detect errors when it can be too late (example: “`TypeError: ...`” in Python)

The Ideal

Well-typed programs cannot go wrong.

Robin Milner: *A Theory of Type Polymorphism in Programming*, JCSS 17(3), 1978

Types

The Ideal

Well-typed programs cannot go wrong.

Robin Milner: *A Theory of Type Polymorphism in Programming*, JCSS 17(3), 1978

Possible failures

1. Corruption of data
2. Null pointer exception
3. Non-termination
4. Out of memory
5. Information leakage
6. ...

Types

The Ideal

Well-typed programs cannot go wrong.

Robin Milner: *A Theory of Type Polymorphism in Programming*, JCSS 17(3), 1978

Possible failures

1. Corruption of data
2. Null pointer exception
3. Non-termination
4. Out of memory
5. Information leakage
6. ...

There are type systems for all of these (and more) aspects but in practice (Java, C#) only (1) is covered.

Type-Related Properties

Type safety

A programming language is **type safe** if the execution of a well-typed program cannot lead to (certain) errors.

Java and the JVM have been proved to be type safe (note: Java exceptions are not errors!)

Types

Type-Related Properties

Type safety

A programming language is **type safe** if the execution of a well-typed program cannot lead to (certain) errors.

Java and the JVM have been proved to be type safe (note: Java exceptions are not errors!)

Soundness of type system

If the type system accepts a program, the semantics does not lead to an error.

Thus: type system must be **justified w.r.t. the semantics**.

Types

Type-Related Properties

Type safety

A programming language is **type safe** if the execution of a well-typed program cannot lead to (certain) errors.

Java and the JVM have been proved to be type safe (note: Java exceptions are not errors!)

Soundness of type system

If the type system accepts a program, the semantics does not lead to an error.

Thus: type system must be **justified w.r.t. the semantics**.

How about **completeness**? Remember Rice's Theorem: Non-trivial semantic properties of programs (such as type-safety of execution) are **undecidable**. Thus automatic analysis of semantic program properties is **necessarily incomplete**.

Typed Semantics

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

Extension of Syntax

- Additional syntactic category: **real numbers** $r \in \mathbb{R}$
(assuming $\mathbb{R} \cap \mathbb{Z} = \emptyset$, i.e., implicit “tagging”)
- **Values** $Val = \mathbb{Z} \uplus \mathbb{R}$ wit $v \in Val$
- **States** $\Sigma = \{\sigma \mid \sigma : Var \rightarrow Val\}$

Extension of Syntax

- Additional syntactic category: **real numbers** $r \in \mathbb{R}$
(assuming $\mathbb{R} \cap \mathbb{Z} = \emptyset$, i.e., implicit “tagging”)
- **Values** $Val = \mathbb{Z} \uplus \mathbb{R}$ wit $v \in Val$
- **States** $\Sigma = \{\sigma \mid \sigma : Var \rightarrow Val\}$

Definition 18.5 (Extended syntax)

$$\begin{aligned} a &::= z \mid r \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp \\ b &::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp \\ c &::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \\ &\quad \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } b \text{ do } c \text{ end} \in Cmd \end{aligned}$$

Typed Evaluation of Arithmetic Expressions

Remember: $a ::= z \mid r \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$

Definition 18.6 (Typed evaluation of arithmetic expressions)

$\rightarrow \subseteq (AExp \times \Sigma) \times Val$ is given by

$$\begin{array}{c} \frac{}{\langle z, \sigma \rangle \rightarrow z} \quad \frac{}{\langle r, \sigma \rangle \rightarrow r} \quad \frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)} \\ \frac{\langle a_1, \sigma \rangle \rightarrow z_1 \quad \langle a_2, \sigma \rangle \rightarrow z_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow z_1 + z_2} \quad \frac{\langle a_1, \sigma \rangle \rightarrow r_1 \quad \langle a_2, \sigma \rangle \rightarrow r_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow r_1 + r_2} \\ \dots \qquad \dots \end{array}$$

Note: **type error** indicated by evaluation getting stuck

Typed Semantics

Typed Evaluation of Arithmetic Expressions

Remember: $a ::= z \mid r \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$

Definition 18.6 (Typed evaluation of arithmetic expressions)

$\rightarrow \subseteq (AExp \times \Sigma) \times Val$ is given by

$$\begin{array}{c} \frac{}{\langle z, \sigma \rangle \rightarrow z} \quad \frac{}{\langle r, \sigma \rangle \rightarrow r} \quad \frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)} \\ \frac{\langle a_1, \sigma \rangle \rightarrow z_1 \quad \langle a_2, \sigma \rangle \rightarrow z_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow z_1 + z_2} \quad \frac{\langle a_1, \sigma \rangle \rightarrow r_1 \quad \langle a_2, \sigma \rangle \rightarrow r_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow r_1 + r_2} \\ \dots \qquad \qquad \qquad \dots \end{array}$$

Note: **type error** indicated by evaluation getting stuck

Example 18.7

If $\sigma(x) = 1 \in \mathbb{Z}$, then $\langle 2 * (x + 1), \sigma \rangle \rightarrow 4$ and $\langle 3.14 * x, \sigma \rangle \not\rightarrow$.

Typed Evaluation of Boolean Expressions

Remember: $b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$

Definition 18.8 (Typed evaluation of Boolean expressions)

$\rightarrow \subseteq (BExp \times \Sigma) \times \mathbb{B}$ is given by

$$\begin{array}{c} \frac{}{\langle t, \sigma \rangle \rightarrow t} \\ \\ \frac{\langle a_1, \sigma \rangle \rightarrow z \quad \langle a_2, \sigma \rangle \rightarrow z}{\langle a_1 = a_2, \sigma \rangle \rightarrow \text{true}} \quad \frac{\langle a_1, \sigma \rangle \rightarrow z_1 \quad \langle a_2, \sigma \rangle \rightarrow z_2}{\langle a_1 = a_2, \sigma \rangle \rightarrow \text{false}} \quad \text{if } z_1 \neq z_2 \\ \\ \frac{\langle a_1, \sigma \rangle \rightarrow r \quad \langle a_2, \sigma \rangle \rightarrow r}{\langle a_1 = a_2, \sigma \rangle \rightarrow \text{true}} \quad \frac{\langle a_1, \sigma \rangle \rightarrow r_1 \quad \langle a_2, \sigma \rangle \rightarrow r_2}{\langle a_1 = a_2, \sigma \rangle \rightarrow \text{false}} \quad \text{if } r_1 \neq r_2 \\ \\ \dots \qquad \qquad \qquad \dots \end{array}$$

Commands: Big- or Small-Step Semantics?

- Need to detect if things “go wrong”
- **Big-step** semantics:
 - cannot model type error by absence of final state – would confuse error and non-termination
 - could introduce extra error element, which would complicate formalisation
- **Small-step** semantics:
 - error = execution gets stuck

Typed Execution of Statements

Definition 18.9 (Typed execution of statements (cf. Definition 16.4))

$\rightarrow_1 \subseteq (Cmd \times \Sigma) \times (Cmd \times \Sigma)$ is given by:

$$\begin{array}{c} \frac{}{\langle \text{skip}, \sigma \rangle \rightarrow_1 \langle \downarrow, \sigma \rangle} \qquad \frac{\langle a, \sigma \rangle \rightarrow v}{\langle x := a, \sigma \rangle \rightarrow_1 \langle \downarrow, \sigma[x \mapsto v] \rangle} \\ \frac{\langle b, \sigma \rangle \rightarrow \text{true}}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end}, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle} \qquad \frac{\langle c_1, \sigma \rangle \rightarrow_1 \langle c'_1, \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \rightarrow_1 \langle c'_1; c_2, \sigma' \rangle} \\ \dots \qquad \dots \end{array}$$

Thus: execution gets stuck iff expression evaluation gets stuck

Typed Execution of Statements

Definition 18.9 (Typed execution of statements (cf. Definition 16.4))

$\rightarrow_1 \subseteq (Cmd \times \Sigma) \times (Cmd \times \Sigma)$ is given by:

$$\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow_1 \langle \downarrow, \sigma \rangle} \qquad \frac{\langle a, \sigma \rangle \rightarrow v}{\langle x := a, \sigma \rangle \rightarrow_1 \langle \downarrow, \sigma[x \mapsto v] \rangle}$$
$$\frac{\langle b, \sigma \rangle \rightarrow \text{true}}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end}, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle} \qquad \frac{\langle c_1, \sigma \rangle \rightarrow_1 \langle c'_1, \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \rightarrow_1 \langle c'_1; c_2, \sigma' \rangle}$$

...

Thus: execution gets stuck iff expression evaluation gets stuck

Example 18.10

For $c = (x := x + 1)$,

- if $\sigma(x) = 2 \in \mathbb{Z}$, then $\langle c, \sigma \rangle \rightarrow_1 \sigma[x \mapsto 3]$
- if $\sigma(x) = 3.14 \in \mathbb{R}$, then $\langle c, \sigma \rangle \not\rightarrow_1$

The Type System

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

The Type System

Data Types and Type Environments

- Data types

$$Typ = \{int, real\}$$

with $\tau \in Typ$

The Type System

Data Types and Type Environments

- Data types

$$Typ = \{\text{int}, \text{real}\}$$

with $\tau \in Typ$

- Observation: type of (arithmetic) expression depends on types of contained variables
- Therefore: introduce **type environments**

$$TEnv = \{\Gamma \mid \Gamma : Var \rightarrow Typ\}$$

The Type System

Data Types and Type Environments

- Data types

$$Typ = \{\text{int}, \text{real}\}$$

with $\tau \in Typ$

- Observation: type of (arithmetic) expression depends on types of contained variables
- Therefore: introduce **type environments**

$$TEnv = \{\Gamma \mid \Gamma : Var \rightarrow Typ\}$$

- In the following: typing rule systems for

- arithmetic expression $a \in AExp$ has type $\tau \in Typ$:
- Boolean expression $b \in BExp$ is well-typed:
- statement $c \in Cmd$ is well-typed:

$$\Gamma \vdash a : \tau$$

$$\Gamma \vdash b$$

$$\Gamma \vdash c$$

in type environment $\Gamma \in TEnv$

The Type System

Typing Rules for Arithmetic Expressions

Remember: $a ::= z \mid r \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$

Definition 18.11 (Well-typedness of arithmetic expressions)

For $\Gamma \in TEnv$, $a \in AExp$, and $\tau \in Typ$, $\Gamma \vdash a : \tau$ is given by

$$\frac{}{\Gamma \vdash z : \text{int}} \quad \frac{}{\Gamma \vdash r : \text{real}} \quad \frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 + a_2 : \tau} \quad \dots$$

The Type System

Typing Rules for Arithmetic Expressions

Remember: $a ::= z \mid r \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$

Definition 18.11 (Well-typedness of arithmetic expressions)

For $\Gamma \in TEnv$, $a \in AExp$, and $\tau \in Typ$, $\Gamma \vdash a : \tau$ is given by

$$\frac{}{\Gamma \vdash z : \text{int}} \quad \frac{}{\Gamma \vdash r : \text{real}} \quad \frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 + a_2 : \tau} \quad \dots$$

Example 18.12 (cf. Example 18.7)

If $\Gamma(x) = \text{int}$, then

$$\frac{\frac{}{\Gamma \vdash 2 : \text{int}} \quad \frac{\Gamma \vdash x : \text{int} \quad \Gamma \vdash 1 : \text{int}}{\Gamma \vdash x + 1 : \text{int}}}{\Gamma \vdash 2 * (x + 1) : \text{int}}$$

and expression $3.14 * x$ is *not* typeable

The Type System

Typing Rules for Boolean Expressions

Remember: $b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$

Definition 18.13 (Well-typedness of Boolean expressions)

For $\Gamma \in TEnv$ and $b \in BExp$, $\Gamma \vdash b$ is given by

$$\frac{}{\Gamma \vdash t} \quad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 = a_2} \quad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 > a_2}$$
$$\frac{\Gamma \vdash b}{\Gamma \vdash \neg b} \quad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \wedge b_2} \quad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \vee b_2}$$

Thus: Boolean expression not well-typed iff some arithmetic subexpression not well-typed or relational operator applied to incompatible arguments

The Type System

Typing Rules for Boolean Expressions

Remember: $b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$

Definition 18.13 (Well-typedness of Boolean expressions)

For $\Gamma \in TEnv$ and $b \in BExp$, $\Gamma \vdash b$ is given by

$$\frac{}{\Gamma \vdash t} \quad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 = a_2} \quad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 > a_2}$$
$$\frac{\Gamma \vdash b}{\Gamma \vdash \neg b} \quad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \wedge b_2} \quad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \vee b_2}$$

Thus: Boolean expression not well-typed iff some arithmetic subexpression not well-typed or relational operator applied to incompatible arguments

Example 18.14

$\Gamma \vdash 2 > 1.0$ does *not* hold

The Type System

Typing Rules for Commands

Definition 18.15 (Well-typedness of commands)

For $\Gamma \in TEnv$ and $c \in Cmd$, $\Gamma \vdash c$ is given by

$$\frac{}{\Gamma \vdash \text{skip}} \quad \frac{\Gamma \vdash a : \Gamma(x)}{\Gamma \vdash x := a} \quad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 ; c_2}$$
$$\frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end}} \quad \frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash \text{while } b \text{ do } c \text{ end}}$$

Thus: command not well-typed iff some expression not well-typed or assignment incompatible

Type Safety

Outline of Lecture 18

Recap: Channel Communication

Fairness in CSP

Summary: Nondeterminism and Parallelism

Types

Typed Semantics

The Type System

Type Safety

Data Types and Type Environments

Observation: even well-typed programs can get stuck if they start in an “unsuitable” state

Example 18.16

If $\Gamma(x) = \text{int}$ and $\sigma(x) = 3.14$, then

$$\Gamma \vdash (x := x + 1) \quad \text{but} \quad \langle x := x + 1, \sigma \rangle \not\rightarrow_1$$

Type Safety

Data Types and Type Environments

Observation: even well-typed programs can get stuck if they start in an “unsuitable” state

Example 18.16

If $\Gamma(x) = \text{int}$ and $\sigma(x) = 3.14$, then

$$\Gamma \vdash (x := x + 1) \quad \text{but} \quad \langle x := x + 1, \sigma \rangle \not\rightarrow_1$$

Definition 18.17 (Well-typedness of states)

Let $\text{type} : \text{Val} \rightarrow \text{Typ}$ be given by $\text{type}(z) := \text{int}$ and $\text{type}(r) := \text{real}$, and let $\Gamma \in \text{TEEnv}$ and $\sigma \in \Sigma$. Then σ is called **well-typed** w.r.t. Γ (notation: $\Gamma \vdash \sigma$) if, for each $x \in \text{Var}$,

$$\text{type}(\sigma(x)) = \Gamma(x)$$

Type Soundness Informally

Type soundness

Execution cannot get stuck: if one runs a well-type command from a well-typed state for a finite number of steps without reaching a final configuration, then (at least) one more step is possible.

Type Safety

Type Soundness Informally

Type soundness

Execution cannot get stuck: if one runs a well-type command from a well-typed state for a finite number of steps without reaching a final configuration, then (at least) one more step is possible.

This property is the combination of **progress** and **preservation**:

Progress

Well-typed programs do not get stuck: if the current configuration is well-typed and not final, then (at least) one more step is possible.

Preservation

Well-typedness is an invariant: the successor of a well-typed configuration is again well-typed.

Type Soundness for Commands

Theorem 18.18 (Progress)

If $c \in \text{Cmd} \setminus \{\downarrow\}$, $\sigma \in \Sigma$ and $\Gamma \in \text{TEnv}$ such that $\Gamma \vdash c$ and $\Gamma \vdash \sigma$, then there exist $c' \in \text{Cmd}$ and $\sigma' \in \Sigma$ such that $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$.

Type Soundness for Commands

Theorem 18.18 (Progress)

If $c \in \text{Cmd} \setminus \{\downarrow\}$, $\sigma \in \Sigma$ and $\Gamma \in \text{TEnv}$ such that $\Gamma \vdash c$ and $\Gamma \vdash \sigma$, then there exist $c' \in \text{Cmd}$ and $\sigma' \in \Sigma$ such that $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$.

Theorem 18.19 (Preservation)

If $c, c' \in \text{Cmd}$, $\sigma, \sigma' \in \Sigma$ and $\Gamma \in \text{TEnv}$ such that $\Gamma \vdash c$, $\Gamma \vdash \sigma$ and $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$, then $\Gamma \vdash c'$ and $\Gamma \vdash \sigma'$.

Type Safety

Type Soundness for Commands

Theorem 18.18 (Progress)

If $c \in \text{Cmd} \setminus \{\downarrow\}$, $\sigma \in \Sigma$ and $\Gamma \in \text{TEnv}$ such that $\Gamma \vdash c$ and $\Gamma \vdash \sigma$, then there exist $c' \in \text{Cmd}$ and $\sigma' \in \Sigma$ such that $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$.

Theorem 18.19 (Preservation)

If $c, c' \in \text{Cmd}$, $\sigma, \sigma' \in \Sigma$ and $\Gamma \in \text{TEnv}$ such that $\Gamma \vdash c$, $\Gamma \vdash \sigma$ and $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$, then $\Gamma \vdash c'$ and $\Gamma \vdash \sigma'$.

Corollary 18.20 (Type soundness)

If $c, c' \in \text{Cmd}$, $\sigma, \sigma' \in \Sigma$ and $\Gamma \in \text{TEnv}$ such that $\Gamma \vdash c$, $\Gamma \vdash \sigma$ and $\langle c, \sigma \rangle \rightarrow_1^* \langle c', \sigma' \rangle$ with $c' \neq \downarrow$, then there exist $c'' \in \text{Cmd}$ and $\sigma'' \in \Sigma$ such that $\langle c', \sigma' \rangle \rightarrow_1 \langle c'', \sigma'' \rangle$.

All proofs by rule induction (based on the corresponding results for expressions)

Type Soundness for Expressions

Lemma 18.21 (Progress for arithmetic expressions)

If $a \in AExp$, $\sigma \in \Sigma$, $\Gamma \in TEnv$ and $\tau \in Typ$ such that $\Gamma \vdash a : \tau$ and $\Gamma \vdash \sigma$, then there exists $v \in Val$ such that $\langle a, \sigma \rangle \rightarrow v$.

Type Safety

Type Soundness for Expressions

Lemma 18.21 (Progress for arithmetic expressions)

If $a \in AExp$, $\sigma \in \Sigma$, $\Gamma \in TEnv$ and $\tau \in Typ$ such that $\Gamma \vdash a : \tau$ and $\Gamma \vdash \sigma$, then there exists $v \in Val$ such that $\langle a, \sigma \rangle \rightarrow v$.

Lemma 18.22 (Preservation for arithmetic expressions)

If $a \in AExp$, $\sigma \in \Sigma$, $\Gamma \in TEnv$, $\tau \in Typ$ and $v \in Val$ such that $\Gamma \vdash a : \tau$ and $\langle a, \sigma \rangle \rightarrow v$, then $\text{type}(v) = \tau$.

Type Safety

Type Soundness for Expressions

Lemma 18.21 (Progress for arithmetic expressions)

If $a \in AExp$, $\sigma \in \Sigma$, $\Gamma \in TEnv$ and $\tau \in Typ$ such that $\Gamma \vdash a : \tau$ and $\Gamma \vdash \sigma$, then there exists $v \in Val$ such that $\langle a, \sigma \rangle \rightarrow v$.

Lemma 18.22 (Preservation for arithmetic expressions)

If $a \in AExp$, $\sigma \in \Sigma$, $\Gamma \in TEnv$, $\tau \in Typ$ and $v \in Val$ such that $\Gamma \vdash a : \tau$ and $\langle a, \sigma \rangle \rightarrow v$, then $\text{type}(v) = \tau$.

Lemma 18.23 (Progress for Boolean expressions)

If $b \in BExp$, $\sigma \in \Sigma$ and $\Gamma \in TEnv$ such that $\Gamma \vdash b$ and $\Gamma \vdash \sigma$, then there exists $t \in \mathbb{B}$ such that $\langle b, \sigma \rangle \rightarrow t$.