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

Syntax of CSP

Definition (Syntax of CSP)

The syntax of CSP is given by

$$
\begin{aligned}
a &::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in \mathit{AExp} \\
b &::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in \mathit{BExp} \\
c &::= \texttt{skip} \mid x := a \mid \alpha?x \mid \alpha!a \mid \\
&\quad\; c_1 ; c_2 \mid \texttt{if}\ \mathit{gc}\ \texttt{fi} \mid \texttt{do}\ \mathit{gc}\ \texttt{od} \mid c_1 \parallel c_2 \in \mathit{Cmd} \\
\mathit{gc} &::= b \to c \mid b \wedge \alpha?x \to c \mid b \wedge \alpha!a \to c \mid \mathit{gc}_1 \,\square\, \mathit{gc}_2 \in \mathit{GCmd}
\end{aligned}
$$

- In $c_1 \parallel c_2$, commands c_1 and c_2 must not use common variables (only local store)
- Guarded command $\mathit{gc}_1 \,\square\, \mathit{gc}_2$ represents an alternative
- In $b \to c$, b acts as a guard that enables the execution of c only if evaluated to true
- $b \wedge \alpha?x \to c$ and $b \wedge \alpha!a \to c$ 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

Semantics of CSP I

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

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \langle \downarrow, \sigma\rangle} \qquad \frac{\langle a, \sigma\rangle \to z}{\langle x := a, \sigma\rangle \to \langle \downarrow, \sigma[x \mapsto z]\rangle}$$

$$\frac{}{\langle \alpha?x, \sigma\rangle \xrightarrow{\alpha?z} \langle \downarrow, \sigma[x \mapsto z]\rangle} \qquad \frac{\langle a, \sigma\rangle \to z}{\langle \alpha!a, \sigma\rangle \xrightarrow{\alpha!z} \langle \downarrow, \sigma\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \xrightarrow{\lambda} \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \xrightarrow{\lambda} \langle c_1'; c_2, \sigma'\rangle} \qquad \frac{\langle gc, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}{\langle \texttt{if } gc \texttt{ fi}, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}$$

$$\frac{\langle gc, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}{\langle \texttt{do } gc \texttt{ od}, \sigma\rangle \xrightarrow{\lambda} \langle c; \texttt{do } gc \texttt{ od}, \sigma'\rangle} \qquad \frac{\langle gc, \sigma\rangle \to \text{fail}}{\langle \texttt{do } gc \texttt{ od}, \sigma\rangle \to \langle \downarrow, \sigma\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \xrightarrow{\lambda} \langle c_1', \sigma'\rangle}{\langle c_1 \parallel c_2, \sigma\rangle \xrightarrow{\lambda} \langle c_1' \parallel c_2, \sigma'\rangle} \qquad \frac{\langle c_2, \sigma\rangle \xrightarrow{\lambda} \langle c_2', \sigma'\rangle}{\langle c_1 \parallel c_2, \sigma\rangle \xrightarrow{\lambda} \langle c_1 \parallel c_2', \sigma'\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \xrightarrow{\alpha?z} \langle c_1', \sigma'\rangle \quad \langle c_2, \sigma\rangle \xrightarrow{\alpha!z} \langle c_2', \sigma\rangle}{\langle c_1 \parallel c_2, \sigma\rangle \to \langle c_1' \parallel c_2', \sigma'\rangle} \qquad \frac{\langle c_1, \sigma\rangle \xrightarrow{\alpha!z} \langle c_1', \sigma\rangle \quad \langle c_2, \sigma\rangle \xrightarrow{\alpha?z} \langle c_2', \sigma'\rangle}{\langle c_1 \parallel c_2, \sigma\rangle \to \langle c_1' \parallel c_2', \sigma'\rangle}$$

Recap: Channel Communication

Semantics of CSP II

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

$$\frac{\langle b, \sigma\rangle \to \text{true}}{\langle b \to c, \sigma\rangle \to \langle c, \sigma\rangle} \qquad \frac{\langle b, \sigma\rangle \to \text{false}}{\langle b \to c, \sigma\rangle \to \text{fail}}$$

$$\frac{\langle b, \sigma\rangle \to \text{true}}{\langle b \wedge \alpha?x \to c, \sigma\rangle \xrightarrow{\alpha?z} \langle c, \sigma[x \mapsto z]\rangle} \qquad \frac{\langle b, \sigma\rangle \to \text{false}}{\langle b \wedge \alpha?x \to c, \sigma\rangle \to \text{fail}}$$

$$\frac{\langle b, \sigma\rangle \to \text{true} \quad \langle a, \sigma\rangle \to z}{\langle b \wedge \alpha!a \to c, \sigma\rangle \xrightarrow{\alpha!z} \langle c, \sigma\rangle} \qquad \frac{\langle b, \sigma\rangle \to \text{false}}{\langle b \wedge \alpha!a \to c, \sigma\rangle \to \text{fail}}$$

$$\frac{\langle gc_1, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}{\langle gc_1 \,\square\, gc_2, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle} \qquad \frac{\langle gc_2, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}{\langle gc_1 \,\square\, gc_2, \sigma\rangle \xrightarrow{\lambda} \langle c, \sigma'\rangle}$$

$$\frac{\langle gc_1, \sigma\rangle \to \text{fail} \quad \langle gc_2, \sigma\rangle \to \text{fail}}{\langle gc_1 \,\square\, gc_2, \sigma\rangle \to \text{fail}}$$

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 $\kappa_0 \to \kappa_1 \to \kappa_2 \to \ldots$ where $\kappa_j = \langle c_1^{(j)} \parallel \ldots \parallel c_n^{(j)}, \sigma_j \rangle$ 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 $\kappa = \langle c_1 \parallel \ldots \parallel c_n, \sigma \rangle$ if there exists $\kappa' = \langle c_1' \parallel \ldots \parallel c_n', \sigma' \rangle$ with $\kappa \to \kappa'$ 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 $\kappa_0 \to \kappa_1 \to \kappa_2 \to \ldots$ where $\kappa_j = \langle c_1^{(j)} \parallel \ldots \parallel c_n^{(j)}, \sigma_j \rangle$ 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. $\langle \texttt{do true} \to x := x + 1 \texttt{ od} \parallel y := y + 1, \ldots \rangle$
   $\to \langle x := x + 1; \texttt{do true} \to x := x + 1 \texttt{ od} \parallel y := y + 1, \ldots \rangle$
   $\to \langle \texttt{do true} \to x := x + 1 \texttt{ od} \parallel y := y + 1, \ldots \rangle \to \ldots$

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

2. $\langle \texttt{do true} \to x := x + 1 \texttt{ od} \parallel \alpha!1 \parallel \alpha?y, \ldots \rangle$
   $\to \langle x := x + 1; \texttt{do true} \to x := x + 1 \texttt{ od} \parallel \alpha!1 \parallel \alpha?y, \ldots \rangle$
   $\to \langle \texttt{do true} \to x := x + 1 \texttt{ od} \parallel \alpha!1 \parallel \alpha?y, \ldots \rangle \to \ldots$

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

3. $\langle \texttt{do } \alpha!1 \to \texttt{skip od} \parallel \texttt{do } \alpha?x \to \texttt{skip od} \parallel \alpha?y, \ldots \rangle$
   $\to \langle \texttt{skip; do } \alpha!1 \to \texttt{skip od} \parallel \texttt{skip; do } \alpha?x \to \texttt{skip od} \parallel \alpha?y, \ldots \rangle$
   $\to \langle \texttt{skip; do } \alpha!1 \to \texttt{skip od} \parallel \texttt{do } \alpha?x \to \texttt{skip od} \parallel \alpha?y, \ldots \rangle$
   $\to \langle \texttt{do } \alpha!1 \to \texttt{skip od} \parallel \texttt{do } \alpha?x \to \texttt{skip od} \parallel \alpha?y, \ldots \rangle \to \ldots$

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

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

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

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!)

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.

Extension of Syntax

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

Definition 18.5 (Extended syntax)

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

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

$$\frac{}{\langle z, \sigma \rangle \rightarrow z} \qquad \frac{}{\langle r, \sigma \rangle \rightarrow r} \qquad \frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow z_1 \quad \langle a_2, \sigma \rangle \rightarrow z_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow z_1 + z_2} \qquad \frac{\langle a_1, \sigma \rangle \rightarrow r_1 \quad \langle a_2, \sigma \rangle \rightarrow r_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow r_1 + r_2}$$

$$\ldots \qquad \ldots$$

Note: type error indicated by evaluation getting stuck

Example 18.7

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

Typed Evaluation of Boolean Expressions

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

Definition 18.8 (Typed evaluation of Boolean expressions)

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

$$\frac{}{\langle t, \sigma \rangle \rightarrow t}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow z \quad \langle a_2, \sigma \rangle \rightarrow z}{\langle a_1{=}a_2, \sigma \rangle \rightarrow \text{true}} \qquad \frac{\langle a_1, \sigma \rangle \rightarrow z_1 \quad \langle a_2, \sigma \rangle \rightarrow z_2}{\langle a_1{=}a_2, \sigma \rangle \rightarrow \text{false}} \quad \text{if } z_1 \neq z_2$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow r \quad \langle a_2, \sigma \rangle \rightarrow r}{\langle a_1{=}a_2, \sigma \rangle \rightarrow \text{true}} \qquad \frac{\langle a_1, \sigma \rangle \rightarrow r_1 \quad \langle a_2, \sigma \rangle \rightarrow r_2}{\langle a_1{=}a_2, \sigma \rangle \rightarrow \text{false}} \quad \text{if } r_1 \neq r_2$$

$\ldots$ $\qquad\qquad$ $\ldots$

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:

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \rightarrow_1 \langle \downarrow, \sigma \rangle} \qquad \frac{\langle a, \sigma \rangle \rightarrow v}{\langle x := a, \sigma \rangle \rightarrow_1 \langle \downarrow, \sigma[x \mapsto v] \rangle}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true}}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle} \qquad \frac{\langle c_1, \sigma \rangle \rightarrow_1 \langle c_1', \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \rightarrow_1 \langle c_1'; c_2, \sigma' \rangle}$$

$$\ldots \qquad \ldots$$

Thus: execution gets stuck iff expression evaluation gets stuck

Example 18.10

For $c = (\texttt{x := x + 1})$,

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

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

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

- In the following: typing rule systems for
  - arithmetic expression $a \in AExp$ has type $\tau \in Typ$: $\Gamma \vdash a : \tau$
  - Boolean expression $b \in BExp$ is well-typed: $\Gamma \vdash b$
  - statement $c \in Cmd$ is well-typed: $\Gamma \vdash c$

  in type environment $\Gamma \in TEnv$

Typing Rules for Arithmetic Expressions

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

Definition 18.11 (Well-typedness of arithmetic expressions)

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

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

Example 18.12 (cf. Example 18.7)

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

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

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

Typing Rules for Boolean Expressions

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

Definition 18.13 (Well-typedness of Boolean expressions)

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

$$\frac{}{\Gamma \vdash t} \qquad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1{=}a_2} \qquad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1{>}a_2}$$

$$\frac{\Gamma \vdash b}{\Gamma \vdash \neg b} \qquad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \wedge b_2} \qquad \frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash b_1 \vee b_2}$$

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

Typing Rules for Commands

Definition 18.15 (Well-typedness of commands)

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

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash a : \Gamma(x)}{\Gamma \vdash x := a} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 \,;\, c_2}$$

$$\frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}} \qquad \frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}}$$

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

Data Types and Type Environments

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

Example 18.16

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

$$\Gamma \vdash (\mathtt{x} := \mathtt{x} + 1) \qquad \text{but} \qquad \langle \mathtt{x} := \mathtt{x} + 1, \sigma \rangle \not\to_1$$

Definition 18.17 (Well-typedness of states)

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

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

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.

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 $c \in Cmd \setminus \{\downarrow\}$, $\sigma \in \Sigma$ *and* $\Gamma \in TEnv$ *such that* $\Gamma \vdash c$ *and* $\Gamma \vdash \sigma$, *then there exist* $c' \in Cmd$ *and* $\sigma' \in \Sigma$ *such that* $\langle c, \sigma \rangle \to_1 \langle c', \sigma' \rangle$.

Theorem 18.19 (Preservation)

If $c, c' \in Cmd$, $\sigma, \sigma' \in \Sigma$ *and* $\Gamma \in TEnv$ *such that* $\Gamma \vdash c$, $\Gamma \vdash \sigma$ *and* $\langle c, \sigma \rangle \to_1 \langle c', \sigma' \rangle$, *then* $\Gamma \vdash c'$ *and* $\Gamma \vdash \sigma'$.

Corollary 18.20 (Type soundness)

If $c, c' \in Cmd$, $\sigma, \sigma' \in \Sigma$ *and* $\Gamma \in TEnv$ *such that* $\Gamma \vdash c$, $\Gamma \vdash \sigma$ *and* $\langle c, \sigma \rangle \to_1^* \langle c', \sigma' \rangle$ *with* $c' \neq \downarrow$, *then there exist* $c'' \in Cmd$ *and* $\sigma'' \in \Sigma$ *such that* $\langle c', \sigma' \rangle \to_1 \langle c'', \sigma'' \rangle$.

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 $\langle a, \sigma \rangle \to v$.

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 $\langle a, \sigma \rangle \to v$, 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 $\langle b, \sigma \rangle \to t$.