Exercises of the lecture on static typing

Programming Paradigms · Session 10 · exercises of the lecture slides · **with solutions**

Exercise $m.n$ is exercise n of the slide "exercises" of block m; exercises 1.8 and 2.5 are stated on the slides "several possible types" and "a remark". Each system is recalled in a box before its exercises. A *derivation* of a judgment is a tree: the judgment stands at the root, every node is the conclusion of an instance of one of the rules, and the node's children are the premises of that instance.

Block 1: the simply typed λ-calculus

The simply typed λ-calculus, with a type annotation on each binder

Terms	$e ::= n \mid b \mid x \mid \lambda x{:}\tau.\, e \mid e\ e \mid e\ op\ e$	n an integer, $b \in \{\texttt{true}, \texttt{false}\}$
Types	$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau \to \tau$	$op \in \{+, <\}$
Judgment	$\Gamma \vdash e : \tau$	Γ a function from variables to types

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \text{(T-Int)} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ \text{(T-Bool)} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \text{(T-Plus)} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}\ \text{(T-Less)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.\, e) : \tau_1 \to \tau_2}\ \text{(T-Abs)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{(T-App)}$$

Conventions. $\Gamma, x{:}\tau$ is Γ in which x now has the type τ (an earlier type of x is hidden); $\emptyset$ is the empty environment, and $x{:}\tau$ alone stands for $\emptyset, x{:}\tau$. The arrow associates to the right: $\texttt{int} \to \texttt{int} \to \texttt{bool}$ is $\texttt{int} \to (\texttt{int} \to \texttt{bool})$. Application associates to the left and binds tighter than $+$, which binds tighter than $<$. The body of an abstraction extends as far to the right as possible.

The calculus without annotations. Its terms are $e ::= n \mid b \mid x \mid \lambda x.\, e \mid e\ e \mid e\ op\ e$, with the same types. Evaluation ignores the annotations: $(\lambda x{:}\tau.\, e)\ v \to e\,[x := v]$.

Exercise 1.1 Two typing derivations

Write typing derivation of

$$\emptyset \vdash (\lambda x{:}\texttt{bool}.\, x)\ \texttt{true} : \texttt{bool} \qquad \text{and} \qquad \emptyset \vdash \lambda x{:}\texttt{int}.\, \lambda y{:}\texttt{int}.\, x < y : \texttt{int} \to \texttt{int} \to \texttt{bool}.$$

Solution. A derivation is built from the root upward: each term form has exactly one rule, so the form of the term decides the rule. In the trees of this sheet the premise $x \in \mathrm{dom}(\Gamma)$ of (T-Var) is checked and not drawn.

First judgment.

$$\dfrac{\dfrac{\dfrac{}{x{:}\texttt{bool} \vdash x : \texttt{bool}}\ \text{(T-Var)}}{\emptyset \vdash \lambda x{:}\texttt{bool}.\, x : \texttt{bool} \to \texttt{bool}}\ \text{(T-Abs)} \qquad \dfrac{}{\emptyset \vdash \texttt{true} : \texttt{bool}}\ \text{(T-Bool)}}{\emptyset \vdash (\lambda x{:}\texttt{bool}.\, x)\ \texttt{true} : \texttt{bool}}\ \text{(T-App)}$$

(T-App) is used with $\tau_1 = \texttt{bool}$ and $\tau_2 = \texttt{bool}$.

Second judgment. The term is $\lambda x{:}\texttt{int}.\, (\lambda y{:}\texttt{int}.\, (x < y))$, since a body extends as far to the right as possible. With $\Gamma = x{:}\texttt{int}, y{:}\texttt{int}$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}}\,\text{(T-Var)} \quad \dfrac{}{\Gamma \vdash y : \texttt{int}}\,\text{(T-Var)}}{x{:}\texttt{int}, y{:}\texttt{int} \vdash x < y : \texttt{bool}}\,\text{(T-Less)}}{x{:}\texttt{int} \vdash \lambda y{:}\texttt{int}.\, x < y : \texttt{int} \to \texttt{bool}}\,\text{(T-Abs)}}{\emptyset \vdash \lambda x{:}\texttt{int}.\, \lambda y{:}\texttt{int}.\, x < y : \texttt{int} \to (\texttt{int} \to \texttt{bool})}\,\text{(T-Abs)}$$

The type at the root is $\texttt{int} \to \texttt{int} \to \texttt{bool}$, the arrow being right-associative.

Remark on the type. The type of the second term cannot be shortened to `bool`: the judgment $\emptyset \vdash \lambda x{:}\texttt{int}.\, \lambda y{:}\texttt{int}.\, x < y : \texttt{bool}$ is not derivable. The term is an abstraction, so a derivation ends with (T-Abs), and the type in the conclusion of (T-Abs) is an arrow type $\tau_1 \to \tau_2$; `bool` is not an arrow type. The type `bool` is the type of the body $x < y$, in the context that gives types to x and y; the type of the function records its two parameters as well.

Exercise 1.2 Two annotated identity functions

Are $(\lambda x{:}\texttt{bool}.\, x)$ and $(\lambda x{:}\texttt{int}.\, x)$ the same programs?

Solution. The answer depends on what "the same" compares: (a) the terms of the annotated syntax, (b) their types in the empty context, (c) the terms once the annotations are removed, and their evaluation.

(a) *No.* The two terms differ in the annotation of the binder, and the annotation is part of the syntax. Renaming the bound variable changes names only, never an annotation, so the terms are not equal up to renaming either.

(b) *No.* By (T-Var) and (T-Abs), $\emptyset \vdash \lambda x{:}\texttt{bool}.\, x : \texttt{bool} \to \texttt{bool}$ and $\emptyset \vdash \lambda x{:}\texttt{int}.\, x : \texttt{int} \to \texttt{int}$. Neither term has the type of the other: a derivation for $\lambda x{:}\texttt{bool}.\, x$ ends with (T-Abs), which gives a type of the form $\texttt{bool} \to \tau_2$, and its premise $x{:}\texttt{bool} \vdash x : \tau_2$ ends with (T-Var), which gives $\tau_2 = \texttt{bool}$. The same argument gives $\texttt{int} \to \texttt{int}$ as the only type of the other term. A client separates the two: $(\lambda x{:}\texttt{int}.\, x)\ 1$ has the type `int`, and $(\lambda x{:}\texttt{bool}.\, x)\ 1$ has no type.

(c) *Yes.* Without the annotations both terms are $\lambda x.\, x$, the identity of session 4. The rule (E-AppAbs) does not consult the annotation, so the two terms also evaluate alike: applied to a value v, each reduces to v in one step.

In summary: the annotation changes which programs the type system accepts, and it changes nothing in what an accepted program computes.

Exercise 1.3 Typing rules without annotations

Redesign typing rules for the λ-calculus without explicit type parameters. What is the type of the identity function $\lambda x.\, x$ now?

Solution. To keep the two systems apart, the solutions write $\Gamma \vdash_{\mathsf{u}} e : \tau$ for the judgment of the calculus without annotations; the statements write it $\Gamma \vdash e : \tau$, the form of the term saying which system is meant.

Six rules keep their form, with $\vdash_{\mathsf{u}}$ in place of $\vdash$: (T-Int), (T-Bool), (T-Var), (T-Plus), (T-Less), (T-App); none of them mentions an annotation. The rule for abstraction changes:

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{u}} e : \tau_2}{\Gamma \vdash_{\mathsf{u}} \lambda x.\, e : \tau_1 \to \tau_2}\,\text{(T-Abs-U)}$$

The premise and the type are as in (T-Abs). The difference is the origin of τ_1: in (T-Abs) the term gives it; in (T-Abs-U) nothing in the term gives it, and the derivation chooses it.

Types of the identity. The types of $\lambda x.\, x$ in the empty context are exactly the types $\tau \to \tau$, one for each type τ.

Each of them is a type of the identity: for any τ,

$$\dfrac{\dfrac{}{x{:}\tau \vdash_{\mathtt{u}} x : \tau}\,\text{(T-Var)}}{\emptyset \vdash_{\mathtt{u}} \lambda x.\, x : \tau \to \tau}\,\text{(T-Abs-U)}$$

There are no others: a derivation of $\emptyset \vdash_{\mathtt{u}} \lambda x.\, x : \sigma$ ends with (T-Abs-U), the only rule for an abstraction, so $\sigma = \tau_1 \to \tau_2$ and the premise is $x{:}\tau_1 \vdash_{\mathtt{u}} x : \tau_2$. That premise ends with (T-Var), the only rule for a variable, which gives the type $(x{:}\tau_1)(x) = \tau_1$; so $\tau_2 = \tau_1$.

Hence $\mathtt{int} \to \mathtt{int}$, $\mathtt{bool} \to \mathtt{bool}$, $(\mathtt{int} \to \mathtt{bool}) \to (\mathtt{int} \to \mathtt{bool})$ are all types of the identity, and $\mathtt{int} \to \mathtt{bool}$ is not. A term no longer has at most one type. The system has no type variables: "$\tau \to \tau$ for every τ" is a family of infinitely many types, described outside the system, and not one type of the system.

Exercise 1.4 A term without annotations

Write typing derivation for $\emptyset \vdash \lambda x.\, \lambda y.\, x\ y : \tau$, for a type τ to find.

Solution. *One derivation,* with $\tau = (\mathtt{int} \to \mathtt{bool}) \to \mathtt{int} \to \mathtt{bool}$ and $\Gamma = x{:}\mathtt{int} \to \mathtt{bool},\ y{:}\mathtt{int}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathtt{u}} x : \mathtt{int} \to \mathtt{bool}}\,\text{(T-Var)} \quad \dfrac{}{\Gamma \vdash_{\mathtt{u}} y : \mathtt{int}}\,\text{(T-Var)}}{x{:}\mathtt{int} \to \mathtt{bool},\ y{:}\mathtt{int} \vdash_{\mathtt{u}} x\ y : \mathtt{bool}}\,\text{(T-App)}}{x{:}\mathtt{int} \to \mathtt{bool} \vdash_{\mathtt{u}} \lambda y.\, x\ y : \mathtt{int} \to \mathtt{bool}}\,\text{(T-Abs-U)}}{\emptyset \vdash_{\mathtt{u}} \lambda x.\, \lambda y.\, x\ y : (\mathtt{int} \to \mathtt{bool}) \to \mathtt{int} \to \mathtt{bool}}\,\text{(T-Abs-U)}$$

All the types. They are exactly the types

$$(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2 \qquad \text{for any types } \tau_1 \text{ and } \tau_2.$$

Each of them is a type of the term: the derivation above stays a derivation when τ_1 replaces $\mathtt{int}$ and τ_2 replaces $\mathtt{bool}$ throughout. There are no others. A derivation of $\emptyset \vdash_{\mathtt{u}} \lambda x.\, \lambda y.\, x\ y : \tau$ ends with two steps (T-Abs-U), so $\tau = \sigma_x \to \sigma_y \to \rho$ and the premise above them is $x{:}\sigma_x, y{:}\sigma_y \vdash_{\mathtt{u}} x\ y : \rho$. That premise ends with (T-App): for some τ_1, the variable x has the type $\tau_1 \to \rho$ and the variable y has the type τ_1. Both are steps (T-Var), and x and y are distinct variables, so $\sigma_x = \tau_1 \to \rho$ and $\sigma_y = \tau_1$. With $\tau_2 = \rho$, the type is $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$.

The shape says what the term does: it takes a function and an argument for it, and returns the result of the function.

Exercise 1.5 Erasure of type annotations

Define type-annotation erasure function $\mathtt{erase}(\cdot)$ and state its correctness theorem.

Solution. *Definition,* by recursion on the annotated term:

$$\begin{aligned} \mathtt{erase}(n) &= n & \mathtt{erase}(\lambda x{:}\tau.\, e) &= \lambda x.\, \mathtt{erase}(e) \\ \mathtt{erase}(b) &= b & \mathtt{erase}(e_1\ e_2) &= \mathtt{erase}(e_1)\ \mathtt{erase}(e_2) \\ \mathtt{erase}(x) &= x & \mathtt{erase}(e_1\ op\ e_2) &= \mathtt{erase}(e_1)\ op\ \mathtt{erase}(e_2) \end{aligned}$$

Only the equation for the abstraction removes something.

Correctness (erasure preserves typing). If $\Gamma \vdash e : \tau$, then $\Gamma \vdash_{\mathtt{u}} \mathtt{erase}(e) : \tau$: same context, same type. The proof is by induction on the derivation of $\Gamma \vdash e : \tau$: erasing every term in the tree turns each step into a step of the same name in the system without annotations, and each step (T-Abs) into a step (T-Abs-U) that chooses for τ_1 the annotation just erased.

Beyond the statement: the converse. The converse as it stands, "if $\Gamma \vdash_{\mathtt{u}} \mathtt{erase}(e) : \tau$ then $\Gamma \vdash e : \tau$", is false. Take $\Gamma = \emptyset$, $e = \lambda x{:}\mathtt{bool}.\, x$ and $\tau = \mathtt{int} \to \mathtt{int}$. Then $\mathtt{erase}(e) = \lambda x.\, x$ and $\emptyset \vdash_{\mathtt{u}} \lambda x.\, x : \mathtt{int} \to \mathtt{int}$ (exercise 1.3), while the only type of $\lambda x{:}\mathtt{bool}.\, x$ is $\mathtt{bool} \to \mathtt{bool}$

(exercise 1.2). The erasure of an untypable term can even be typable: $\lambda x{:}\texttt{bool}.\, x + 1$ has no type, because (T-Plus) asks for $x{:}\texttt{bool} \vdash x : \texttt{int}$; its erasure $\lambda x.\, x + 1$ has the type $\texttt{int} \to \texttt{int}$.

A converse that holds. If $\Gamma \vdash_{\mathsf{u}} e' : \tau$ for a term e' without annotations, then *there exists* an annotated term e with $\texttt{erase}(e) = e'$ and $\Gamma \vdash e : \tau$. The proof is again by induction on the derivation: each step (T-Abs-U) has chosen a type τ_1 for its bound variable, and e is e' with that type written on the binder. The annotations depend on the derivation, not on e' alone: the two derivations of exercise 1.3 for $\texttt{int} \to \texttt{int}$ and $\texttt{bool} \to \texttt{bool}$ give the two terms of exercise 1.2.

Together: the terms typable without annotations are exactly the erasures of the typable annotated terms.

Exercise 1.6 Typability in the empty context

What does it mean for a program e to be well-typed in an empty context, i.e. $\emptyset \vdash e : \tau$ for some type τ?

Solution. The judgment says two things, (a) about the free variables of e and (b) about its evaluation under call by value.

(a) *The term e is closed: it has no free variable.* More generally, if $\Gamma \vdash e : \tau$ then every free variable of e is in $\mathrm{dom}(\Gamma)$. The proof is by induction on the derivation. (T-Var) demands $x \in \mathrm{dom}(\Gamma)$. (T-Abs) types the body in $\Gamma, x{:}\tau_1$: the free variables of the body are in $\mathrm{dom}(\Gamma) \cup \{x\}$, and those of the abstraction, which exclude x, are in $\mathrm{dom}(\Gamma)$. The other rules pass Γ unchanged to subterms whose free variables together are those of the term. With $\Gamma = \emptyset$, the set of free variables is empty: e is a complete program, which needs no value from outside.

(b) *The evaluation of e never gets stuck.* A term is stuck when it is not a value (a constant or an abstraction) and no reduction rule applies to it, as for $1 + \texttt{true}$ or $1\ 2$. Safety has two parts. *Progress:* a closed term with a type is a value or takes a step. *Preservation:* if $\emptyset \vdash e : \tau$ and e takes a step to e', then $\emptyset \vdash e' : \tau$. By the two together, every term reached from e is closed and has the type τ, so it is a value or steps again. An evaluation that stops therefore stops at a value of type τ: an integer constant when $\tau = \texttt{int}$, $\texttt{true}$ or $\texttt{false}$ when $\tau = \texttt{bool}$, an abstraction when τ is an arrow type.

The judgment promises nothing more: not that the value is the intended one, and nothing in a non-empty context, where the term is not a program ($x{:}\texttt{int} \vdash x + 1 : \texttt{int}$ is derivable, and $x + 1$ cannot be evaluated without a value for x). Safety by itself does not say that evaluation stops; in this calculus it always does, by a separate theorem (normalization), which the argument above does not use.

Exercise 1.7 Self-application

Is there any context Γ and type τ such that $\Gamma \vdash x\ x : \tau$?

Solution. *No context and no type do.* Suppose a derivation of $\Gamma \vdash x\ x : \tau$ exists.

1. The term is an application, so the derivation ends with (T-App): for some type τ_1,
$$\Gamma \vdash x : \tau_1 \to \tau \qquad \text{and} \qquad \Gamma \vdash x : \tau_1$$
are both derivable.
2. The only rule for a variable is (T-Var). If $x \notin \mathrm{dom}(\Gamma)$, its premise fails and neither judgment is derivable: the supposition is already refuted. If $x \in \mathrm{dom}(\Gamma)$, the rule gives x the single type $\Gamma(x)$, because Γ is a function. So
$$\Gamma(x) = \tau_1 \to \tau \qquad \text{and} \qquad \Gamma(x) = \tau_1, \qquad \text{hence} \qquad \tau_1 = \tau_1 \to \tau.$$
3. No type satisfies this equation. Let the size of a type be its number of symbols:
$$|\texttt{int}| = |\texttt{bool}| = 1 \qquad \text{and} \qquad |\sigma \to \sigma'| = 1 + |\sigma| + |\sigma'|.$$

Every type is a finite expression, so its size is a positive integer, and the size of $\tau_1 \to \tau$ is $1 + |\tau_1| + |\tau|$, greater than $|\tau_1|$. Two types of different sizes are different.

Each step was forced, so no derivation exists, for any Γ and τ. The argument uses only (T-App) and (T-Var), which are the same in the calculus without annotations: $x\ x$ has no type there either. A derivation for a term contains a derivation for each of its subterms, so no term with $x\ x$ as a subterm has a type. The term of session 4 whose evaluation never ends,

$$\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x),$$

is therefore rejected, with any annotations.

Block 1: mini-ML with monomorphic typing

Mini-ML. A function $\texttt{fun}\ x \to e$ carries no annotation, and a primitive stands before its argument: $+(x, 1)$.

$$\begin{array}{lll} \text{Expressions} & e ::= x \mid c \mid op \mid \texttt{fun}\ x \to e \mid e\ e \mid (e, e) \mid \texttt{let}\ x = e\ \texttt{in}\ e \\ & c \text{ a constant } (1, 2, \ldots, \mathit{true}, \ldots), & op \text{ a primitive } (+, \times, \mathit{fst}, \ldots) \\ \text{Types} & \tau ::= \texttt{int} \mid \texttt{bool} \mid \cdots \mid \tau \to \tau \mid \tau \times \tau \end{array}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ (\textsc{Var}) \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ (\textsc{Const})\ \ldots \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}\ (\textsc{Prim})\ \ldots$$

$$\frac{\Gamma + x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2}\ (\textsc{Fun}) \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\ (\textsc{App})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ (\textsc{Let})$$

$\Gamma + x{:}\tau$ is the environment Γ' defined by $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ otherwise. The slides give these rules without names; the names are added here, to label the steps of a derivation.

Exercise 1.8 An identity function applied to itself

Show that the expression

$$((\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x))\ \ 42$$

is typable.

Solution. We proceed from the leaves:

1. The inner identity $\texttt{fun}\ x \to x$ can be given the type $\texttt{int} \to \texttt{int}$:
$$\frac{x : \texttt{int} \vdash x : \texttt{int}}{\emptyset \vdash \texttt{fun}\ x \to x\ :\ \texttt{int} \to \texttt{int}}$$
2. The outer identity is applied to a value of type $\texttt{int} \to \texttt{int}$, so we choose its type as $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$:
$$\frac{x : \texttt{int} \to \texttt{int} \ \vdash\ x : \texttt{int} \to \texttt{int}}{\emptyset \vdash \texttt{fun}\ x \to x\ :\ (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})}$$
3. Apply the outer to the inner:
$$\frac{\emptyset \vdash \texttt{fun}\ x \to x\ :\ (\texttt{int}{\to}\texttt{int}) \to (\texttt{int}{\to}\texttt{int}) \qquad \emptyset \vdash \texttt{fun}\ x \to x\ :\ \texttt{int}{\to}\texttt{int}}{\emptyset \vdash (\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x)\ :\ \texttt{int} \to \texttt{int}}$$
4. Apply the result to 42:
$$\frac{\emptyset \vdash (\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x)\ :\ \texttt{int} \to \texttt{int} \qquad \emptyset \vdash 42 : \texttt{int}}{\emptyset \vdash ((\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x))\ 42\ :\ \texttt{int}}$$

The key insight is that the *two occurrences* of $\texttt{fun}\ x \to x$ receive *different* monomorphic types. This is possible because they are syntactically separate sub-expressions.

Block 2: the typing of the While language

The While language (written WHILE on the slides)

Expressions	$e ::= c \mid x \mid e\ op\ e$
	c an integer or Boolean constant, op a binary operator $(+, <, \dots)$
Statements	$s ::= x \leftarrow e \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ e\ \texttt{do}\ s \mid s;\ s \mid \texttt{skip}$
Types	$\tau ::= \texttt{int} \mid \texttt{bool}$

Typing of expressions: $\Gamma \vdash e : \tau$, "in environment Γ, expression e has type τ".

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \text{(T-Int)} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ \text{(T-Bool)} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \text{(T-Plus)} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}\ \text{(T-Less)} \qquad \text{etc.}$$

Type checking of statements: $\Gamma \vdash s$, "in environment Γ, statement s is well-typed".

$$\frac{}{\Gamma \vdash \texttt{skip}}\ \text{(T-Skip)} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2}\ \text{(T-Seq)} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e}\ \text{(T-Assign)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}\ \text{(T-If)} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s}\ \text{(T-While)}$$

The slides give these rules without names; the names are added here, to label the steps of a derivation.

Consider a small extension of the WHILE language with

a) new expressions: pair constructors (e, e) and get the left and the right pair components written $\texttt{fst}\, e$ and $\texttt{snd}\, e$ respectively;
b) new statements: $\texttt{for}\ x = e\ \texttt{to}\ e\ \texttt{do}\ s$ and $\texttt{repeat}\ s\ \texttt{until}\ e$ loops.

Exercise 2.1 Pair types and the typing of pairs

Define the pair types. Define typing rules for each of the pair (e_1, e_2), $\texttt{fst}\, e$ and $\texttt{snd}\, e$ expressions.

Solution.

τ	$::=$	**type**
	$\mid\ \texttt{int}$	type of integer values
	$\mid\ \texttt{bool}$	type of Boolean values
	$\mid\ (\tau \times \tau)$	pair type

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\, e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\, e : \tau_2}$$

Exercise 2.2 A rejected program

Consider the program

$$\texttt{if}\ \texttt{fst}\, x > 0\ \texttt{then}\ x \leftarrow (\texttt{snd}\, x, \texttt{fst}\, x + \texttt{snd}\, x)\ \texttt{else}\ x \leftarrow 0$$

Explain why this program is rejected by the static type system.

Solution. For $\texttt{fst}\, x$ to be well-typed, x must have a pair type, say $\Gamma(x) = \texttt{int} \times \texttt{int}$. Then in the `then` branch, the assignment $x \leftarrow (\texttt{snd}\, x, \texttt{fst}\, x + \texttt{snd}\, x)$ assigns a value of type $\texttt{int} \times \texttt{int}$ to x, which is consistent. However, in the `else` branch, the assignment $x \leftarrow 0$ assigns a value of type `int` to x. Since $\Gamma(x) = \texttt{int} \times \texttt{int}$, the typing rule for assignment requires the right-hand side to have the same type as x, but $\texttt{int} \neq \texttt{int} \times \texttt{int}$. Therefore, the program is rejected.

(Alternatively, if we assume $\Gamma(x) = \texttt{int}$, then $\texttt{fst}\, x$ is not well-typed. There is no type for x that makes the whole program type-check.)

Exercise 2.3 The typing rule of the repeat loop

Define the typing rule for the repeat loop $\texttt{repeat}\ s\ \texttt{until}\ e$.

Solution.

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{repeat}\ s\ \texttt{until}\ e}$$

Exercise 2.4 The typing rule of the for loop

Define the typing rule for the for loop $\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ s$.

Solution.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \Gamma, i : \texttt{int} \vdash s}{\Gamma \vdash \texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ s}$$

Note that the loop variable i is bound in the body s with type `int`, which extends the typing environment.

Exercise 2.5 The conditional as a statement and as an expression

The WHILE language has the conditional at the level of statements (the rule on the left); a conditional could be added in the expressions (the rule on the right):

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

What is different and what is common in both cases?

Solution. *Common.*

- The condition is an expression of the type `bool`, typed in the same Γ as the conditional: its value selects a branch, and only `true` and `false` select one.
- Both branches are checked, in the same Γ, although a run takes one of them only. The check is static: it does not know the value of the condition, so it covers both.
- Each rule is the only one for its form, and its premises are judgments on the three parts of the conditional and on nothing else.

Different.

- The judgments differ. A statement is well-typed or not, $\Gamma \vdash s$, and has no type: it produces no value and acts by changing the environment. An expression has a type, $\Gamma \vdash e : \tau$, the type of the value it produces.
- Hence the constraint between the branches. The rule of the statement asks only that each branch is well-typed; nothing relates the two. The rule of the expression asks that both branches have *the same type* τ, and that type is the type of the conditional: the value of the conditional is the value of one branch, which one is not known statically, and the context of the conditional needs one type for it.
- The price is the rejection of expressions whose every run is fine: $\texttt{if true then } 1 \texttt{ else false}$

has the value 1, under the evaluation rule that takes the first branch when the condition is `true`, and has no type, since (T-Int) gives 1 the type `int` and (T-Bool) gives `false` the type `bool`. The statement `if true then` $x \leftarrow 1$ `else` $y \leftarrow$ `false` is well-typed in $\{x \mapsto \texttt{int}, y \mapsto \texttt{bool}\}$: its branches assign different variables, of different types.