SW7 Programming Paradigms

Static Type Systems

block 2: typing of an imperative language

Léon Gondelman

AALBORG UNIVERSITY | COPENHAGEN | 2026

statements and state: a program with effects

a statement has **no value**: its run **changes a state**

the state is an **environment** E: the **value of each variable**

```
a ← 1;
b ← a + 1;
a ← a + b
```

before line 1: $E = \emptyset$

the semantics relates a statement and an environment to a **new environment**:

$$\emptyset,\ \ a \leftarrow 1;\ b \leftarrow a + 1;\ a \leftarrow a + b \ \ \twoheadrightarrow \ \ \{a \mapsto 3,\ b \mapsto 2\}$$

statements and state: a program with effects

a statement has **no value**: its run **changes a state**

the state is an **environment** E: the **value of each variable**

```
▶ a ← 1;
  b ← a + 1;
  a ← a + b
```

after line 1: $E = \{a \mapsto 1\}$

the semantics relates a statement and an environment to a **new environment**:

$$\emptyset,\ \ a \leftarrow 1;\ b \leftarrow a + 1;\ a \leftarrow a + b \ \ \twoheadrightarrow \ \ \{a \mapsto 3,\ b \mapsto 2\}$$

statements and state: a program with effects

a statement has **no value**: its run **changes a state**

the state is an **environment** E: the **value of each variable**

```
  a ← 1;
▶ b ← a + 1;
  a ← a + b
```

after line 2: $E = \{a \mapsto 1,\ b \mapsto 2\}$

the semantics relates a statement and an environment to a **new environment**:

$$\emptyset,\ \ a \leftarrow 1;\ b \leftarrow a + 1;\ a \leftarrow a + b \ \ \twoheadrightarrow \ \ \{a \mapsto 3,\ b \mapsto 2\}$$

statements and state: a program with effects

a statement has **no value**: its run **changes a state**

the state is an **environment** E: the **value of each variable**

```
  a ← 1;
  b ← a + 1;
▶ a ← a + b
```

after line 3: $E = \{a \mapsto 3,\ b \mapsto 2\}$

the semantics relates a statement and an environment to a **new environment**:

$$\emptyset,\ \ a \leftarrow 1;\ b \leftarrow a + 1;\ a \leftarrow a + b \ \ \twoheadrightarrow \ \ \{a \mapsto 3,\ b \mapsto 2\}$$

typing with effects: one type for each variable

what does line 3 add 1 to?

a Boolean **in the runs where** $y < 0$, an integer in the other runs

a typing environment Γ gives **one type** to each variable, for the **whole program**

```
x ← 1;
if y < 0 then x ← true else skip;
z ← x + 1
```

an assignment $x \leftarrow e$ is accepted only if e has **the type of** x: the **value** of a variable changes, its **type** does not

typing with effects: one type for each variable

what does line 3 add 1 to?

a Boolean **in the runs where** $y < 0$, an integer in the other runs

a typing environment Γ gives **one type** to each variable, for the **whole program**

```
  x ← 1;
▶ if y < 0 then x ← true else skip;
  z ← x + 1
```

with $\Gamma = \{x \mapsto \texttt{int},\ y \mapsto \texttt{int},\ z \mapsto \texttt{int}\}$:
line 2 is **rejected**, since `true` is not of type `int`

an assignment $x \leftarrow e$ is accepted only if e has **the type of** x: the **value** of a variable changes, its **type** does not

big-step and small-step semantics

small-step semantics (block 1): one judgment for **one step**; a run is a sequence of steps

$$1 + 2 < 4 \;\to\; 3 < 4 \;\to\; \texttt{true}$$

big-step semantics, or natural semantics: one judgment for a **whole run**, from the program to its result

$$\emptyset,\; a \leftarrow 1 + 2 \;\twoheadrightarrow\; \{a \mapsto 3\}$$

runs without a result, in the two semantics

statement, run from $\emptyset$	big-step	small-step
$a \leftarrow 1 + 2$	$E' = \{a \mapsto 3\}$	steps, then the run ends
$a \leftarrow 1 + \texttt{true}$	**no** E'	**stuck**: not finished, and no step applies
`while true do skip`	**no** E'	**one more step**, forever

what is the result of a run that crashes, and of a run that never ends?

big-step gives **no result** to both; small-step tells them apart

block 2 uses **big-step**: one rule for each construct, read as an interpreter

Typing of an imperative language

1. the WHILE language and its big-step semantics
2. the typing of expressions and statements
3. type safety of WHILE, stated formally
4. the lab: semantics and typing of `break` and `continue`

a WHILE program

```
a ← 0;
b ← 1;
while b < 100 do
  b ← a + b;
  a ← b - a
```

what does this program compute?

a WHILE program

```
a ← 0;
b ← 1;
while b < 100 do
  b ← a + b;
  a ← b - a
```

what does this program compute?

a and b are two **consecutive Fibonacci numbers**

the run ends in the environment

$$\{a \mapsto 89,\ b \mapsto 144\}$$

syntax of the WHILE language

e	$::=$	**expression**
	c	integer or Boolean constant
$\mid$	x	variable
$\mid$	$e\ op\ e$	binary operator ($+$, $<$, …)
s	$::=$	**statement**
$\mid$	$x \leftarrow e$	assignment
$\mid$	`if` e `then` s `else` s	conditional
$\mid$	`while` e `do` s	loop
$\mid$	$s;\ s$	sequence
$\mid$	`skip`	do nothing

big-step operational semantics of WHILE

we seek to define a relation between some expression e and a **value** v

$$e \twoheadrightarrow v$$

here, values are limited to integer and Boolean constants

v	$::=$		**value**
	$\mid$	n	integer
	$\mid$	b	Boolean

caveat: in most languages, values do not coincide with constants (Java: appendix)

value of an expression in an environment

in the environment $E = \{a \mapsto 34,\ b \mapsto 55\}$, the expression $a + b$ has the value 89:

$$E,\ a + b \ \twoheadrightarrow\ 89$$

an **environment** E is a function from variables to values: it gives **the value of each variable**

the relation $E, e \twoheadrightarrow v$ reads "in the environment E, the expression e **has the value** v"

semantics of expressions

the relation $E, e \twoheadrightarrow v$ is defined by **inference rules**:

$$\frac{}{E, n \twoheadrightarrow n}\ (\textsc{Int}) \qquad \frac{}{E, b \twoheadrightarrow b}\ (\textsc{Bool}) \qquad \frac{x \in \mathrm{dom}(E)}{E, x \twoheadrightarrow E(x)}\ (\textsc{Var})$$

$$\frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n = n_1 + n_2}{E, e_1 + e_2 \twoheadrightarrow n}\ (\textsc{Plus})$$

- each of the other operators has **a rule of the form of (Plus)**
- (Plus) applies to **integers** only: $1 + \texttt{true}$ has no rule, and no value

derivation of the value of an expression (example)

with $E = \{a \mapsto 34,\ b \mapsto 55\}$, we have

$$\dfrac{\dfrac{a \in \mathrm{dom}(E)}{E, a \twoheadrightarrow 34} \qquad \dfrac{b \in \mathrm{dom}(E)}{E, b \twoheadrightarrow 55} \qquad 89 = 34 + 55}{E, a + b \twoheadrightarrow 89}$$

such a tree is a **proof**: here, that the expression has the value 89

semantics of statements

a statement **changes the value** of some variables, through assignments

its semantics is therefore a relation between **two environments**:

$$E, s \twoheadrightarrow E'$$

it reads “in the environment E, the run of the statement s **ends**, in the environment E'”

semantics of statements: skip, assignment, sequence

$$\frac{}{E, \texttt{skip} \twoheadrightarrow E}\ (\textsc{Skip}) \qquad \frac{E, e \twoheadrightarrow v}{E, x \leftarrow e \twoheadrightarrow E\{x \mapsto v\}}\ (\textsc{Assign})$$

$$\frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, s_2 \twoheadrightarrow E_2}{E, s_1; s_2 \twoheadrightarrow E_2}\ (\textsc{Seq})$$

- $E\{x \mapsto v\}$ is E in which x **now has the value** v
- in (Seq), s_2 runs in E_1, the **environment that** s_1 **leaves**

$\{a \mapsto 1\}, \quad a \leftarrow a + 1;\ a \leftarrow a \times 2 \quad \twoheadrightarrow \quad \{a \mapsto 4\}$ by (Seq), with $E_1 = \{a \mapsto 2\}$

semantics of statements: conditional

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s_1 \twoheadrightarrow E_1}{E, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_1}\ (\textsc{IfTrue})$$

$$\frac{E, e \twoheadrightarrow \texttt{false} \quad E, s_2 \twoheadrightarrow E_2}{E, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_2}\ (\textsc{IfFalse})$$

- **one branch only** stands in the premises: the other branch does not run
- a guard whose value is not a Boolean has **no rule**: the conditional has no result

$$\{a \mapsto 21\},\ \ \texttt{if}\ a > 0\ \texttt{then}\ a \leftarrow 2 \times a\ \texttt{else skip} \ \twoheadrightarrow\ \{a \mapsto 42\} \qquad \text{by } (\textsc{IfTrue})$$

semantics of statements: loop

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E}\ (\textsc{WhileFalse})$$

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E_2}{E, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E_2}\ (\textsc{WhileTrue})$$

- the last premise of (WhileTrue) is about **the same loop**, in the environment E_1 that the body leaves
- a loop that never ends has **no derivation**: a derivation is a finite tree

$\{a \mapsto 0\}$, $\texttt{while}\ a < 2\ \texttt{do}\ a \leftarrow a + 1 \twoheadrightarrow \{a \mapsto 2\}$ by (WhileTrue) twice, then (WhileFalse)

derivation of the run of a conditional (example)

with $E = \{a \mapsto 21\}$, we have

$$\dfrac{\dfrac{E, a \twoheadrightarrow 21 \quad E, 0 \twoheadrightarrow 0}{E, a > 0 \twoheadrightarrow \texttt{true}} \qquad \dfrac{\dfrac{E, 2 \twoheadrightarrow 2 \quad E, a \twoheadrightarrow 21}{E, 2 \times a \twoheadrightarrow 42}\,(\textsc{Assign})}{E, a \leftarrow 2 \times a \twoheadrightarrow \{a \mapsto 42\}}}{E, \texttt{if}\ a > 0\ \texttt{then}\ a \leftarrow 2 \times a\ \texttt{else skip} \twoheadrightarrow \{a \mapsto 42\}}\,(\textsc{IfTrue})$$

the premises $21 > 0$ and $42 = 2 \times 21$ of the two operator rules are not drawn

expressions without a value, statements without a result

for some expressions e, there is **no value** v such that $E, e \twoheadrightarrow v$

$1 + \texttt{true}$	no rule adds a Boolean

for some statements s, there is **no environment** E' such that $E, s \twoheadrightarrow E'$

`while true do skip`	the run **never ends**
`while` $(42 + 0)$ `do skip`	the value of the guard is **not a Boolean**
`while` $(42/0)$ `do skip`	the guard has **no value**

Typing of WHILE

1. the WHILE language and its big-step semantics
2. **the typing of expressions and statements**
3. type safety of WHILE, stated formally
4. the lab: semantics and typing of `break` and `continue`

types of WHILE

we introduce **types**, with the following abstract syntax

τ ::=		**type**
	`int`	integer values
	`bool`	Boolean values

typing of expressions (the rules of block 1)

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

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

with $\Gamma = \{a \mapsto \texttt{int}\}$: $\Gamma \vdash a + 1 < 5 : \texttt{bool}$ holds; $a + \texttt{true}$ has **no type**

typing derivation of an expression (example)

with $\Gamma = \{a \mapsto \texttt{int}\}$, we have

$$\dfrac{\dfrac{a \in \mathrm{dom}(\Gamma)}{\Gamma \vdash a : \texttt{int}} \qquad \dfrac{}{\Gamma \vdash 1 : \texttt{int}}}{\Gamma \vdash a + 1 : \texttt{int}}$$

this derivation is a proof that `a+1` is a **well-typed expression**

expressions without a type

with $\Gamma = \{a \mapsto \texttt{int}\}$, two expressions have **no type**:

b	$b \notin \mathrm{dom}(\Gamma)$: (T-VAR) does not apply
$1 + \texttt{true}$	(T-PLUS) requires $\Gamma \vdash \texttt{true} : \texttt{int}$

is anything lost?

no: $1 + \texttt{true}$ has **no value** in any environment, and b has none where only a has a value

the safety theorem of this block states the link between **having a type** and **having a value**

typing judgment for statements

a statement has no value, hence **no type**: it is **well typed**, or it is not

the judgment for statements is

$$\Gamma \vdash s$$

it reads “in the typing environment Γ, the statement s is **well typed**”

typing of statements: skip, sequence, assignment

$$\frac{}{\Gamma \vdash \texttt{skip}}\ \text{(T-Skip)} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2}\ \text{(T-Seq)}$$

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e}\ \text{(T-Assign)}$$

- in (T-Assign), the **same type** τ stands in both premises: e has the type of x
- Γ is **the same** in premises and conclusion: a statement changes E, never Γ

with $\Gamma = \{a \mapsto \texttt{int}\}$: $\Gamma \vdash a \leftarrow a + 1$ **but not** $\Gamma \vdash a \leftarrow \texttt{true}$: `true` is not of type `int`

typing of statements: conditional and loop

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}\ \text{(T-IF)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ s}\ \text{(T-WHILE)}$$

- **both branches** are checked, where a run executes one of them
- the body of a loop is checked **once**, where a run executes it any number of times

with $\Gamma = \{a \mapsto \texttt{int}\}$: $\Gamma \vdash \texttt{while}\ a < 100\ \texttt{do}\ a \leftarrow a + a$ **but not** $\Gamma \vdash \texttt{while}\ a\ \texttt{do skip}$

the conditional as a statement and as an expression

we have conditional at the level of statements:

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

we could add conditional in the expressions:

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

what is different, and what is common, in the two rules?

exercises (1/2): typing of pairs

an extension of WHILE with pairs: a constructor (e, e), and the left and right components $\texttt{fst}\, e$ and $\texttt{snd}\, e$

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

2. Consider the program

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

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

exercises (PDF)

exercises (2/2): typing of two more loops

an extension of WHILE with two statements: `repeat` s `until` e and `for` $x = e$ `to` e `do` s

3. Define the typing rule for the repeat loop `repeat` s `until` e.

4. Define the typing rule for the for loop `for` $i = e_1$ `to` e_2 `do` s.

exercises (PDF)

Type safety of WHILE

1. the WHILE language and its big-step semantics
2. the typing of expressions and statements
3. **type safety of WHILE, stated formally**
4. the lab: semantics and typing of `break` and `continue`

type safety: the slogan of block 1 (recall)

well-typed programs cannot “go wrong”

- what does it mean formally?

- why does it matter in practice?

- how is type safety enforced?

type safety as progress and preservation

Safety = Progress + Preservation

Progress: a well-typed program is **not stuck**: either it is a final result or it can take a step of evaluation

Preservation: if a well-typed program takes a step of evaluation, then the resulting program is **also well-typed**

a formal statement needs a mathematical definition of “step of evaluation”, “final result” and “well-typed”

the need for a formal definition

a type system designed without a proof of safety can be flawed

- Java and Scala: programs accepted by the type checker that fail with a cast error, found in 2016 (Amin and Tate, OOPSLA 2016)
- TypeScript: soundness is not a design goal

a proof of safety needs three definitions, each one precise:

- the **syntax**: what a program is
- the **semantics**: what a program does, and what “going wrong” is
- the **typing rules**: which programs are accepted

agreement of an environment with a typing environment

with $\Gamma = \{a \mapsto \texttt{int},\ b \mapsto \texttt{bool}\}$:

$E_1 = \{a \mapsto 34,\ b \mapsto \texttt{true}\}$	**agrees** with Γ
$E_2 = \{a \mapsto 34,\ b \mapsto \texttt{true},\ c \mapsto 7\}$	**agrees** with Γ: Γ says nothing of c
$E_3 = \{a \mapsto \texttt{true},\ b \mapsto \texttt{true}\}$	does **not** agree: the value of a is not an integer
$E_4 = \{a \mapsto 34\}$	does **not** agree: b has no value

definition

an environment E **agrees** with a typing environment Γ, written $\Gamma \vdash E$, when for every variable $x \in \mathrm{dom}(\Gamma)$:

- $x \in \mathrm{dom}(E)$, and
- $E(x)$ is a value of type $\Gamma(x)$: an integer if $\Gamma(x) = \texttt{int}$, a Boolean if $\Gamma(x) = \texttt{bool}$

type safety of WHILE: the theorem for expressions

theorem (safety, expressions)

if $\Gamma \vdash e : \tau$ and $\Gamma \vdash E$, then **there is a value** v such that $E, e \twoheadrightarrow v$, and v is **of type** τ

a well-typed expression **has a value**, and the value is of its type

with $\Gamma = \{a \mapsto \texttt{int}\}$ and $E = \{a \mapsto 21\}$: $\Gamma \vdash 2 \times a : \texttt{int}$, and $E,\ 2 \times a \twoheadrightarrow 42$, an integer

the operators are $+$, $-$, $\times$, $<$, $>$, $=$ on integers, **each defined for all integers**

type safety of WHILE: the theorem for statements

theorem (safety, statements)

if $\Gamma \vdash s$ and $\Gamma \vdash E$ and $E, s \twoheadrightarrow E'$, then $\Gamma \vdash E'$

a well-typed statement **preserves the agreement**: when its run ends, every variable of Γ still has a value of its type

with $\Gamma = \{a \mapsto \texttt{int}\}$ and $E = \{a \mapsto 21\}$:

$$E,\ \texttt{if}\ a > 0\ \texttt{then}\ a \leftarrow 2 \times a\ \texttt{else skip} \twoheadrightarrow \{a \mapsto 42\} \qquad \text{and} \qquad \Gamma \vdash \{a \mapsto 42\}$$

limits of the safety theorem: a run that never ends

does every well-typed statement have a result?

this statement is **well typed**, with $\Gamma = \emptyset$

its run never ends: **no E' exists**

the theorem on statements covers only **the runs that end**

```
while true do skip
```

limits of the safety theorem: division by zero

$42/0$ has the type `int`, with $/$ typed as $+$, and it has **no value**

with division, the theorem on expressions **fails**

the type checkers of OCaml and Java accept this division too: the error is raised **at run time**

```
x ← 42 / 0
```

limits of the safety theorem: progress

progress: “a well-typed program is never stuck”

progress speaks of **steps**: the judgment $E, s \twoheadrightarrow E'$ **cannot state it**

a remedy within big-step semantics: the unwanted end of a run receives **a result of its own**

the theorem then states that **no well-typed program ends with it**

the lab does so for a `break` or a `continue` outside a loop

The lab: `break` and `continue`

1. the WHILE language and its big-step semantics
2. the typing of expressions and statements
3. type safety of WHILE, stated formally
4. **the lab: semantics and typing of `break` and `continue`**

static typing as a static analysis

a **static analysis** establishes a property of **all the runs** of a program from its text alone, **without running it**

- a variable read before it is assigned
- code that no run reaches
- two threads that write the same variable without a lock (session 7)
- an operation applied to a value of the wrong type

each of these properties is **undecidable** in general (by reduction from the halting problem), so an analysis that always answers and never misses an error **rejects some correct programs**

a type system can also check an **effect** of a statement, not only the type of a value: in the lab, that `break` and `continue` occur only inside a loop

lab: big-step semantics of `break` and `continue`

two new statements:

$$s ::= \cdots \mid \texttt{break} \mid \texttt{continue}$$

the run of a statement now ends with an **outcome**:

$$\omega ::= \texttt{normal} \mid \texttt{broken} \mid \texttt{continued}$$

the judgment for statements becomes

$$E, s \twoheadrightarrow (E', \omega)$$

```
x ← 1;
break;
x ← 2
```

in which environment, and with which outcome, does the run from $\{x \mapsto 0\}$ end?

to write: the rules of `break` and `continue`, **three** rules for $s_1; s_2$, **four** for `while` e `do` s

lab: typing of `break` and `continue`

the judgment for statements carries a **flag**:

$$(\Gamma, \ell) \vdash s \qquad \ell \in \{\texttt{true}, \texttt{false}\}$$

ℓ records whether s stands **inside a loop**

a whole program is checked with $\ell = \texttt{false}$

to write: the typing rules of `break`, `continue` and `while` e `do` s

```
if x > 0 then break else skip
```

```
while true do
  if x > 0 then break else skip
```

which of the two do the rules accept, with $(\{x \mapsto \texttt{int}\}, \texttt{false})$*?*

the property that the typing guarantees

if $(\Gamma, \texttt{false}) \vdash s$ and $E, s \twoheadrightarrow (E', \omega)$, then $\omega = \texttt{normal}$

Appendix

big-step rules: expressions, skip, assignment, sequence

$$\frac{}{E, n \twoheadrightarrow n}\ (\textsc{Int}) \qquad \frac{}{E, b \twoheadrightarrow b}\ (\textsc{Bool}) \qquad \frac{x \in \mathrm{dom}(E)}{E, x \twoheadrightarrow E(x)}\ (\textsc{Var})$$

$$\frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n = n_1 + n_2}{E, e_1 + e_2 \twoheadrightarrow n}\ (\textsc{Plus})$$

likewise for the other operators

$$\frac{}{E, \texttt{skip} \twoheadrightarrow E}\ (\textsc{Skip}) \qquad \frac{E, e \twoheadrightarrow v}{E, x \leftarrow e \twoheadrightarrow E\{x \mapsto v\}}\ (\textsc{Assign}) \qquad \frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, s_2 \twoheadrightarrow E_2}{E, s_1; s_2 \twoheadrightarrow E_2}\ (\textsc{Seq})$$

big-step rules: conditional and loop

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s_1 \twoheadrightarrow E_1}{E, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_1}\ (\textsc{IfTrue}) \qquad \frac{E, e \twoheadrightarrow \texttt{false} \quad E, s_2 \twoheadrightarrow E_2}{E, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_2}\ (\textsc{IfFalse})$$

$$\frac{E, e \twoheadrightarrow \texttt{false}}{E, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E}\ (\textsc{WhileFalse})$$

$$\frac{E, e \twoheadrightarrow \texttt{true} \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E_2}{E, \texttt{while}\ e\ \texttt{do}\ s \twoheadrightarrow E_2}\ (\textsc{WhileTrue})$$

typing of WHILE: all the rules

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

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

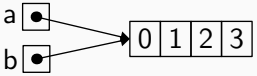
$$\frac{}{\Gamma \vdash \texttt{skip}}\ \text{(T-Skip)} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2}\ \text{(T-Seq)} \qquad \frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e}\ \text{(T-Assign)}$$

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

values and addresses in Java (example)

in Java (or Python, OCaml, etc.), a value may be an address, even if we do not have addresses among the literal constants of the language

```
int[] a = new int[4];
...
int[] b = a;
b[2] = 42;
...
```



values and addresses in Java (example)

in Java (or Python, OCaml, etc.), a value may be an address, even if we do not have addresses among the literal constants of the language

```
int[] a = new int[4];
...
int[] b = a;
b[2] = 42;
...
```

