$\langle exp \rangle ::= \langle var \rangle \mid \langle exp \rangle\ \langle exp \rangle \mid \lambda\langle var \rangle.\ \langle exp \rangle$

This is what you expect. Constants and usual operations for booleans and integers

$\mid 0 \mid 1 \mid 2 \mid \cdots \mid -\langle exp \rangle \mid \langle exp \rangle \ \{+, -, \times, \div, \text{rem}\}\ \langle exp \rangle$

$\mid \text{true} \mid \text{false} \mid \neg\langle exp \rangle \mid \langle exp \rangle \ \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}\ \langle exp \rangle$

$\mid \text{if } \langle exp \rangle \text{ then } \langle exp \rangle \text{ else } \langle exp \rangle \mid \langle exp \rangle \ \{=, \neq, <, \leq, >, \geq\}\ \langle exp \rangle$

constructing a tuple

$\mid \langle \langle exp \rangle, \cdots, \langle exp \rangle \rangle \mid \langle exp \rangle.\langle tag \rangle$ — projection: destructing a tuple

constructing an alternative

$\mid @\langle tag \rangle\ \langle exp \rangle \mid \text{sumcase } \langle exp \rangle \text{ of } (\langle exp \rangle, \cdots, \langle exp \rangle)$ — case analysis for destructing an alternative.

$\mid \text{error} \mid \text{typeerror}$

error: other kind of error

typeerror

We already said that introducing four new kinds of canonical forms : amounts to introducing four new runtime (or dynamic) types to the language, namely, int, bool, tuple and alternative. For tuple and alternative, we have operations for constructing tuple and alternative expressions, and also operations for destructing them. This is a typical pattern that appears repeatedly when one designs a new programming language. When she or he adds a static or dynamic/runtime type to a language, he or she introduces appropriate constructors and destructors (for the type) to the language.

⑤ Evaluation relation $\Rightarrow$ should also be extended to incorporate intended semantics of newly added operations. For integers and booleans, we change $\Rightarrow$ according to the standard semantics of primitive operations. Here we give rules for only some operations. Other cases are similar.

$$\frac{e \Rightarrow \bar{i}}{-e \Rightarrow \hat{-}\bar{i}}$$

$\bar{i}$: integer constant. $-e$: symbol. $\hat{-}$: mathematical operation. I use $\hat{}$ to emphasize it.

$$\frac{e \Rightarrow b}{\neg e \Rightarrow \hat{\neg} b}$$

b: boolean constant. $\neg e$: symbol.

$$\frac{e \Rightarrow \bar{i} \qquad e' \Rightarrow \bar{i}'}{e \ op \ e' \Rightarrow \bar{i} \ \hat{op} \ \bar{i}'}$$

where $op \in \{+, -, \times, =, \neq, <, \leq, >, \geq\}$ or $(op \in \{\div, \text{rem}\}$ and $\bar{i}' \neq 0)$

$$\frac{e \Rightarrow b \qquad e' \Rightarrow b'}{e \ op \ e' \Rightarrow b \ \hat{op} \ b'}$$

when $op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

$$\frac{e \Rightarrow \text{true} \qquad e' \Rightarrow z}{\text{if } e \text{ then } e' \text{ else } e'' \Rightarrow z} \qquad \frac{e \Rightarrow \text{false} \qquad e'' \Rightarrow z}{\text{if } e \text{ then } e' \text{ else } e'' \Rightarrow z}$$

For operators for tuples and alternatives, we include rules for constructors that just evaluate their arguments, and those for destructors that convert constructed tuples and alternatives back to some of their components.

$$\frac{e_0 \Rightarrow z_0 \quad \cdots \quad e_{n-1} \Rightarrow z_{n-1}}{\langle e_0, \cdots, e_{n-1} \rangle \Rightarrow \langle z_0, \cdots, z_{n-1} \rangle}$$

$$\frac{e \Rightarrow \langle z_0, \cdots, z_{n-1} \rangle}{e.k \Rightarrow z_k} \quad \text{when } k < n.$$