Expression Types: Parentheses and Constants

**Expanded Typing Rules**

- **Parentheses**
  \[
  \Gamma \vdash e_1 : \tau_2 \\
  \tau_2 = \tau_1 \\
  \Gamma \vdash e_2 : \tau_3 \\
  \Gamma \vdash e_3 : \tau_4 \\
  \tau_2 = \text{bool} \\
  \tau_3 = \tau_1 \\
  \tau_4 = \tau_1 \\
  \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_1
  \]

- **Type Inferencing Rules**
  \[
  \Gamma \vdash e_1 : \tau_2 \\
  \tau_2 = \text{bool} \\
  \Gamma \vdash e_2 : \tau_3 \\
  \tau_3 = \tau_1 \\
  \Gamma \vdash e_1 \text{ andalso } e_2 : \tau_1
  \]

**Boolean Operators: andalso**

- andalso
  \[
  \Gamma \vdash e_1 : \tau_2 \\
  \Gamma \vdash e_2 : \tau_3 \\
  \tau_1 = \text{bool} \\
  \tau_2 = \text{bool} \\
  \tau_3 = \text{bool} \\
  \Gamma \vdash e_1 \text{ andalso } e_2 : \tau_1
  \]

**Boolean Operators: orelse**

- orelse
  \[
  \Gamma \vdash e_1 : \tau_2 \\
  \Gamma \vdash e_2 : \tau_3 \\
  \tau_1 = \text{bool} \\
  \tau_2 = \text{bool} \\
  \Gamma \vdash e_1 \text{ orelse } e_2 : \tau_1
  \]

**If Expressions**

- if
  \[
  \Gamma \vdash e_1 : \tau_2 \\
  \Gamma \vdash e_2 : \tau_3 \\
  \tau_1 = \text{bool} \\
  \tau_2 = \text{bool} \\
  \tau_3 = \tau_1 \\
  \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_1
  \]
Monadic Operator Application Rule

Monadic Operator Application
\[ \text{monop}(\oplus) = (\forall \alpha_1 \ldots \alpha_n . \tau \rightarrow \tau') \]
\[ \Gamma \vdash e : \tau'' \quad \Gamma \vdash e' : \tau''' \]
\[ \Gamma \vdash \sigma : \tau' \]
\[ \Gamma \vdash \sigma_2 : \tau'' \]
\[ \sigma_2 = \text{unify}((\{\sigma_1(\tau'' \rightarrow \tau'''), \sigma_1(\tau'' \rightarrow \tau''')\} + \sigma_2)) \]
\[ \Gamma \vdash \sigma_3 : \tau''' \]
\[ \Gamma \vdash (e_1, e_2) : \tau'' | \sigma_2 \]

Binary Operator Application Rule

Binary Operator Application
\[ \text{binop}(\oplus) = (\forall \alpha_1 \ldots \alpha_n . \tau \rightarrow \tau' \rightarrow \tau'') \]
\[ \Gamma \vdash e : \tau'' \quad \Gamma \vdash e' : \tau''' \]
\[ \Gamma \vdash \sigma_2 : \tau'' \]
\[ \sigma_2 = \text{unify}((\{\sigma_2 \circ \sigma_1(\tau'' \rightarrow \tau'''') \circ \sigma_2 \circ \sigma_1(\tau'' \rightarrow \tau''')\} + \sigma_2)) \]
\[ \Gamma \vdash (e_1 \oplus e_2, m) : \tau''' | \sigma_3 \]

Fun Expressions

Fun Expressions
\[ \Gamma \vdash e : \tau_1 | \sigma_1 \]
\[ \Gamma \vdash e_1 : \tau_2 | \sigma_1 \]
\[ \sigma_2 = \text{unify}((\{\sigma_2 \circ \sigma_1(\tau_2 \rightarrow \tau_1)\} + \sigma_2)) \]
\[ \Gamma \vdash (e_1, e_2) : \tau_1 | \sigma_3 \]

Declaration Rules: Single Declarations

Val Declaration
\[ \Gamma \vdash e : \tau_1 \]
\[ \Gamma \vdash \text{val} \ x = e : \{\tau_2 \rightarrow \text{Gen}(\tau_1,\tau_3)\} | \sigma_1 \]
\[ \Gamma \vdash \text{let} \ d \ in \ e \ : \ \tau_1 \]

Recursive Declarations
\[ \Gamma \vdash \{f \rightarrow \tau_1\} + \{x \rightarrow \tau_2\} + e : \tau_3 \]
\[ \Gamma \vdash \text{fun} \ f \ x = e : \{f \rightarrow \text{Gen}(\tau_1)\} | \sigma_1 \]

\[ \Gamma \vdash \text{let} \ d \ in \ e \ : \ \tau_1 | \sigma_2 \]
Logic Programming Evaluation Background

- Programs are simple logical formulae
  - Programs are sets of Horn Clauses, a subset of predicate logic
  - Computation (in the pure subset) is logical deduction
  - Clauses viewed as axioms and primitive inference rules
- Programs are "declarative": no imperative features; no sense of reassigning
  - In program, all variables have implicit scope limited to each clause

Logic Programming Evaluation Semantics

\[
\text{Prog} \vdash \sigma
\]

\[
\begin{align*}
\text{(Term' := Body')} & \in \text{Prog} \\
V & = \text{free_vars(Term')} \cup \text{free_vars(Body')} \\
\sigma_1 & = \text{fresh}(V, \text{Term' := Body'}) \\
\sigma_2 & = \text{unify}(\text{Term}, \sigma_1(\text{Term'})) \\
\text{Prog} \vdash \sigma_2 \circ \sigma_1(\text{Body'}) & \quad \sigma_3(\text{Body'}) \quad \sigma_2 \circ \sigma_1
\end{align*}
\]