MicroML: Minimal Functional Programming Language, Expressions

\[
\begin{align*}
\langle \text{exp} \rangle & \ ::= \quad \text{IDENT} \mid \text{BOOL} \mid \text{INT} \mid \text{REAL} \mid \text{STRING} \mid ( ) \mid [] \mid (\langle \text{exp} \rangle) \\
& \quad \langle \text{exp} \rangle \ \text{andalso} \ \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \ \text{orelse} \ \langle \text{exp} \rangle \\
& \quad \text{let} \ \langle \text{dec} \rangle \ \text{in} \ \langle \text{exp} \rangle \ \text{end} \\
& \quad \text{if} \ \langle \text{exp} \rangle \ \text{then} \ \langle \text{exp} \rangle \ \text{else} \ \langle \text{exp} \rangle \\
& \quad \text{fn} \ \text{IDENT} \Rightarrow \ \langle \text{exp} \rangle \\
& \quad \langle \text{exp} \rangle \ \langle \text{infid} \rangle \ \langle \text{exp} \rangle \mid \langle \text{prefid} \rangle \ \langle \text{exp} \rangle \\
& \quad \langle \text{exp} \rangle \ \langle \text{exp} \rangle \\
\langle \text{dec} \rangle & \ ::= \quad \text{val} \ \text{IDENT} \ = \ \langle \text{exp} \rangle \\
& \quad \text{fun} \ \text{IDENT} \ \text{IDENT}^{+} \ = \langle \text{exp} \rangle \\
& \quad \langle \text{dec} \rangle \langle \text{dec} \rangle \\
& \quad \text{local} \ \langle \text{dec} \rangle \ \text{in} \ \langle \text{dec} \rangle \ \text{end}
\end{align*}
\]
TYIDENT refers to a type variable name.

In MicroML type system, will associate types with expressions, and typing environments (mapping expression variables to types)
Type Variables and Structural Polymorphism

- Many functions don’t need complete type information to work
- Examples:
  - Swapping elements in a pair
  - Counting elements in a list
- Point: only care about outer structure; want types to reflect that
- Idea: Use type variables to represent “don’t need the type”
  - Still can capture “but different components must have the same type”
- In mid-term, may need “Here I don’t care about the type, but I need to share it with you” (monomorphism)
- Might know “The type can be anything and I don’t share it with anybody” (structural polymorphism)
- Will record polymorphic types for expression variables
- Will derive monomorphic types for expressions
Type system background

- Greek letters $\alpha, \beta, \gamma, \alpha_i, \ldots$ are mathematical variables that represent type variables.
- Greek letters $\tau, \tau_i, \tau', \ldots$ used as mathematical variables that represent monomorphic types.
- $\Gamma, \Delta$ range over typing environments (:\text{IDENT} \rightarrow\text{< polyty >})
- Expression constants and operators have a “best” polymorphic type given by \text{const\_sig}
- Expression typing judgments look like $\Gamma \vdash e : \tau$
- Declaration typing judgment looks like $\Gamma \vdash \text{dec} : \Delta$
- Greek letters $\sigma, \sigma_i, \sigma'$ are mathematical variables that represent simultaneous substitutions (:\text{< ty >} \rightarrow\text{< ty >}).
Type Substitution

- Substitutions replace variables by terms
- Often need to do so in a variety of related (even hierarchical) contexts
- Often use the same notation (overloaded) for substitutions in all related contexts (really are different functions).
- Start: $TYIDENT \rightarrow \langle ty \rangle$
  - Often represent substitutions by association lists:
    $(TYIDENT \times \langle ty \rangle) list$
  - Empty list represents map that takes each type variable to itself
    $(\alpha, \tau) \# l \equiv l[\alpha \mapsto \tau]$
  - Will write $\tau[\tau_1/\alpha_1, \ldots \tau_n/\alpha_n]$ for $[(\alpha_1, \tau_1), \ldots (\alpha_n, \tau_n)](\tau)$
- Next: “Lift” to $\langle ty \rangle$
Lifting Type Substitution

“Lift” to $<\text{ty}>$:
- $\text{lift}(\sigma)(\alpha) = \sigma(\alpha)$ and as $<\text{ty}>$
- $\text{lift}(\sigma)(<\text{tyconstant}>) = <\text{tyconstant}>$
- $\text{lift}(\sigma)((\tau)) = \text{lift}(\sigma)(\tau)$
- $\text{lift}(\sigma)((\tau_1,\ldots,\tau_n)<\text{tyconstructor}>)) = ((\text{lift}(\sigma)(\tau_1),\ldots,(\text{lift}(\sigma)(\tau_n)))<\text{tyconstructor}>))$
- $\text{lift}(\sigma)(\tau \rightarrow \tau') = (\text{lift}(\sigma)(\tau)) \rightarrow (\text{lift}(\sigma)(\tau'))$
- $\text{lift}(\sigma)(\tau \ast \tau') = (\text{lift}(\sigma)(\tau)) \ast (\text{lift}(\sigma)(\tau'))$

Defined in layers recursively BUT defines a single simultaneous substitution

Will write $\sigma(\tau)$ for $\text{lift}(\sigma)(\tau)$
Free Type Variables

- A type variable is a **free type variable** in a monomorphic type if it occurs in it
  - \(\text{free}_\text{vars}(\alpha) = \{\alpha\}\) and as \(<\text{ty}>\)
  - \(\text{free}_\text{vars}(<\text{tyconstant}>) = \{\}\)
  - \(\text{free}_\text{vars}((\tau)) = \text{free}_\text{vars}(\tau)\)
  - \(\text{free}_\text{vars}((\tau_1, \ldots, \tau_n) <\text{tyconstructor}>)) = \text{free}_\text{vars}(\tau_1) \cup \ldots \cup \text{free}_\text{vars}(\tau_n)\)
  - \(\text{free}_\text{vars}(\tau \rightarrow \tau') = \text{free}_\text{vars}(\tau) \cup \text{free}_\text{vars}(\tau')\)
  - \(\text{free}_\text{vars}(\tau \star \tau') = \text{free}_\text{vars}(\tau) \cup \text{free}_\text{vars}(\tau')\)
  - \(\text{free}_\text{vars}(\forall \alpha_1 \ldots \alpha_n . \tau) = \text{free}_\text{vars}(\tau) - \{\alpha_1, \ldots, \alpha_n\}\)
  - \(\text{free}_\text{vars}(\Gamma) = \bigcup_{\tau \in \text{range}(\Gamma)} \text{free}_\text{vars}(\tau)\)