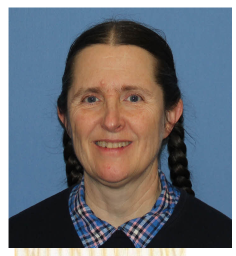
Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC



https://courses.engr.illinois.edu/cs421/sp2023

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Example

- Which rule do we apply?

$$\frac{?}{\{\}\ |\text{-}\ (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \text{ -> } (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}}$$

Example

- Let rec rule: 2 {one : int list} |- (let x = 2 in fun y -> (x :: y :: one)) : int → int list

1 {one : int list} |- (1 :: one) : int list

$$\frac{\{one : int\ list\} \vdash (1 :: one) : int\ list \qquad \{one : int\ list\} \vdash (let\ x = 2\ in\ fun\ y \rightarrow (x :: y :: one)) : int \rightarrow int\ list}{\{\} \vdash (let\ rec\ one = 1 :: one\ in\ let\ x = 2\ in\ fun\ y \rightarrow (x :: y :: one)) : int \rightarrow int\ list}$$

{} |- (let rec one = 1 :: one in
let x = 2 in
fun y -> (x :: y :: one)) : int → int list

Proof of 1

- Which rule?

$$\frac{}{\{one : int\ list\} \mid\!- (1 :: one) : int\ list}$$

Proof of 1

- Binary Operator

3 4

$$\frac{\{one : int\ list\} \mid\!- \ 1 : int \qquad \{one : int\ list\} \mid\!- \ one : int\ list}{\{one : int\ list\} \mid\!- \ (1 :: one) : int\ list}$$

where $(::) : int \rightarrow int\ list \rightarrow int\ list$

Proof of 1

3

Constant Rule

$$\frac{}{\{one : int\ list\} \mid\!- \ 1 : int}$$

4

Variable Rule

$$\frac{}{\{one : int\ list\} \mid\!- \ one : int\ list}$$

$$\frac{\{one : int\ list\} \mid\!- \ 1 : int \qquad \{one : int\ list\} \mid\!- \ one : int\ list}{\{one : int\ list\} \mid\!- \ (1 :: one) : int\ list}$$

Proof of 2

- Let Rule

$$\frac{\{one : int\ list\} \mid\!- 2{:}int \qquad \begin{array}{l}\{x{:}int;\ one : int\ list\} \mid\!- \\ fun\ y \to \\ \quad (x :: y :: one)) \\ : int \rightarrow int\ list\end{array}}{\begin{array}{l}\{one : int\ list\} \mid\!- (let\ x = 2\ in \\ \quad fun\ y \to (x :: y :: one)) : int \rightarrow int\ list\end{array}}$$

Proof of 2

5 {x:int; one : int list} |- fun y -> (x :: y :: one)) : int → int list

- Constant

$$\frac{\dfrac{}{\{one : int\ list\} \mid\!- 2:int} \qquad \{x:int;\ one : int\ list\} \mid\!- fun\ y \to (x :: y :: one)) : int \to int\ list}{\{one : int\ list\} \mid\!- (let\ x = 2\ in\ fun\ y \to (x :: y :: one)) : int \to int\ list}$$

Proof of 5

$$\frac{?}{\{x{:}int;\ one : int\ list\}\ |\text{-}\ fun\ y\ \text{->}\ (x :: y :: one))\ : int \rightarrow int\ list}$$

Proof of 5

$$\frac{\dfrac{?}{\{y{:}int;\ x{:}int;\ one : int\ list\} \ |\!-\ (x :: y :: one) : int\ list}}{\{x{:}int;\ one : int\ list\} \ |\!-\ fun\ y \ {-}{>}\ (x :: y :: one)) : int \rightarrow int\ list}$$

By the Fun Rule

Proof of 5

$$\dfrac{\overset{⑥}{\dfrac{?}{\{y{:}int;\ x{:}int;\ one{:}int\ list\}\ |\!-\ x{:}int}} \qquad \overset{⑦}{\dfrac{?}{\{y{:}int;\ x{:}int;\ one{:}int\ list\}\ |\!-\ (y :: one) : int\ list}}}{\dfrac{\{y{:}int;\ x{:}int;\ one : int\ list\}\ |\!-\ (x :: y :: one) : int\ list}{\{x{:}int;\ one : int\ list\}\ |\!-\ \text{fun } y \text{ -> } (x :: y :: one)) : int \rightarrow int\ list}}$$

By BinOp where $(::) : int \rightarrow int\ list \rightarrow int\ list$

Proof of 6

6

Variable Rule

{y:int; x:int; one:int list}
|- x:int

7

?

{y:int; x:int; one:int list}
|- (y :: one) : int list

{y:int; x:int; one : int list} |- (x :: y :: one) : int list

{x:int; one : int list} |- fun y -> (x :: y :: one))
: int → int list

Proof of 7

- Binary Operation Rule

$$
\dfrac{\dfrac{?}{\{y{:}int;\ \ldots\}\ |\text{-}\ y{:}int} \qquad \dfrac{\dfrac{?}{\{\ldots;\ one{:}int\ list;\ldots\}}}{|\text{-}\ one : int\ list}}{\{y{:}int;\ x{:}int;\ one : int\ list\}\ |\text{-}\ (y :: one) : int\ list}
$$

By BinOp where $(::) : int \rightarrow int\ list \rightarrow int\ list$

Proof of 7

$$\dfrac{\dfrac{\text{Variable Rule}}{\{y{:}\text{int};\ \ldots\}\ |\!-\ y{:}\text{int}} \qquad \dfrac{\dfrac{\text{Variable Rule}}{\{\ldots;\ \text{one}{:}\text{int list};\ldots\}}}{|\!-\ \text{one} : \text{int list}}}{\{y{:}\text{int};\ x{:}\text{int};\ \text{one} : \text{int list}\}\ |\!-\ (y :: \text{one}) : \text{int list}}$$

Curry - Howard Isomorphism

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms

- Function space arrow corresponds to implication; application corresponds to modus ponens

Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

- Application

$$\frac{\Gamma \;|\!\!-\; e_1 : \alpha \rightarrow \beta \quad \Gamma \;|\!\!-\; e_2 : \alpha}{\Gamma \;|\!\!-\; (e_1\; e_2) : \beta}$$

Mea Culpa

- The above system can't handle polymorphism as in OCAML
- No type variables in type language (only meta-variables in the logic)
- Would need:
  - Object level type variables and some kind of type quantification
  - **let** and **let rec** rules to introduce polymorphism
  - Explicit changes to rules to eliminate (instantiate) polymorphism

Support for Polymorphic Types

- Monomorpic Types (τ):
  - Basic Types: int, bool, float, string, unit, …
  - Type Variables: α, β, γ, δ, ε
  - Compound Types: $\alpha \rightarrow \beta$, int * string, bool list, …
- Polymorphic Types:
  - Monomorphic types τ
  - Universally quantified monomorphic types
  - $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$
  - Can think of τ as same as $\forall \,.\, \tau$

Support for Polymorphic Types

- Typing Environment Γ supplies polymorphic types (which will often just be monomorphic) for variables
- Free variables of monomorphic type just type variables that occur in it
  - Write FreeVars(τ)
- Free variables of polymorphic type removes variables that are universally quantified
  - $\text{FreeVars}(\forall\alpha_1, \ldots, \alpha_n . \tau) = \text{FreeVars}(\tau) - \{\alpha_1, \ldots, \alpha_n\}$
- FreeVars(Γ) = all FreeVars of types in range of Γ

Example FreeVars Calculations

- Vars(‘a -> (int -> ‘b) -> ‘a) = {‘a , ‘b}
- FreeVars (All ‘b. ‘a -> (int -> ‘b) -> ‘a) =

  {‘a , ‘b} – {‘b} = {‘a}
- FreeVars {x : All ‘b. ‘a -> (int -> ‘b) -> ‘a,

  id: All ‘c. ‘c -> ‘c,

  y: All ‘c. ‘a -> ‘b -> ‘c} =

  {‘a} U {} U {‘a, ‘b} = {‘a, ‘b}