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?

?

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

Example

- Let rec rule:

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

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

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

Proof of 1

- Binary Operator

③ {one : int list} |- 1: int

④ {one : int list} |- one : int list

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

where (::) : int → int list → int list

Proof of 1

③ Constant Rule {one : int list} |- 1: int

④ Variable Rule {one : int list} |- one : int list

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

Proof of 2

- Let Rule

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

{one : int list} |- 2:int

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

Proof of 2

- Constant

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

{one : int list} |- 2:int

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

Proof of 5

?

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

Proof of 5

?

{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

By the Fun Rule

Proof of 5

⑥ ?

{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

By BinOp where (::) : int → int list → 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

$$\frac{\dfrac{?}{\{y{:}int;\ \ldots\} \vdash y{:}int} \qquad \dfrac{?}{\{\ldots;\ one{:}int\ list;\ldots\} \vdash one : int\ list}}{\{y{:}int;\ x{:}int;\ one : int\ list\} \vdash (y :: one) : int\ list}$$

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

Proof of 7

$$\frac{\dfrac{\text{Variable Rule}}{\{y{:}int;\ \ldots\} \vdash y{:}int} \qquad \dfrac{\text{Variable Rule}}{\{\ldots;\ one{:}int\ list;\ldots\} \vdash one : int\ list}}{\{y{:}int;\ x{:}int;\ one : int\ list\} \vdash (y :: one) : int\ list}$$

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

$$\frac{\Gamma \vdash e_1 : \alpha \rightarrow \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash (e_1\ e_2) : \beta}$$

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
  - FreeVars($\forall\alpha_1, \ldots, \alpha_n . \tau$) = 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}