

Programming Languages and Compilers (CS 421)



Elsa L Gunter

2112 SC, UIUC

<http://courses.engr.illinois.edu/cs421>

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



Nested Recursive Types

```
# type 'a labeled_tree =  
  TreeNode of ('a * 'a labeled_tree  
  list);;
```

```
type 'a labeled_tree = TreeNode of ('a  
  * 'a labeled_tree list)
```



Nested Recursive Type Values

```
# let ltree =  
  TreeNode(5,  
    [TreeNode (3, []);  
      TreeNode (2, [TreeNode (1, []);  
                          TreeNode (7, [])]);  
      TreeNode (5, [])]);;
```

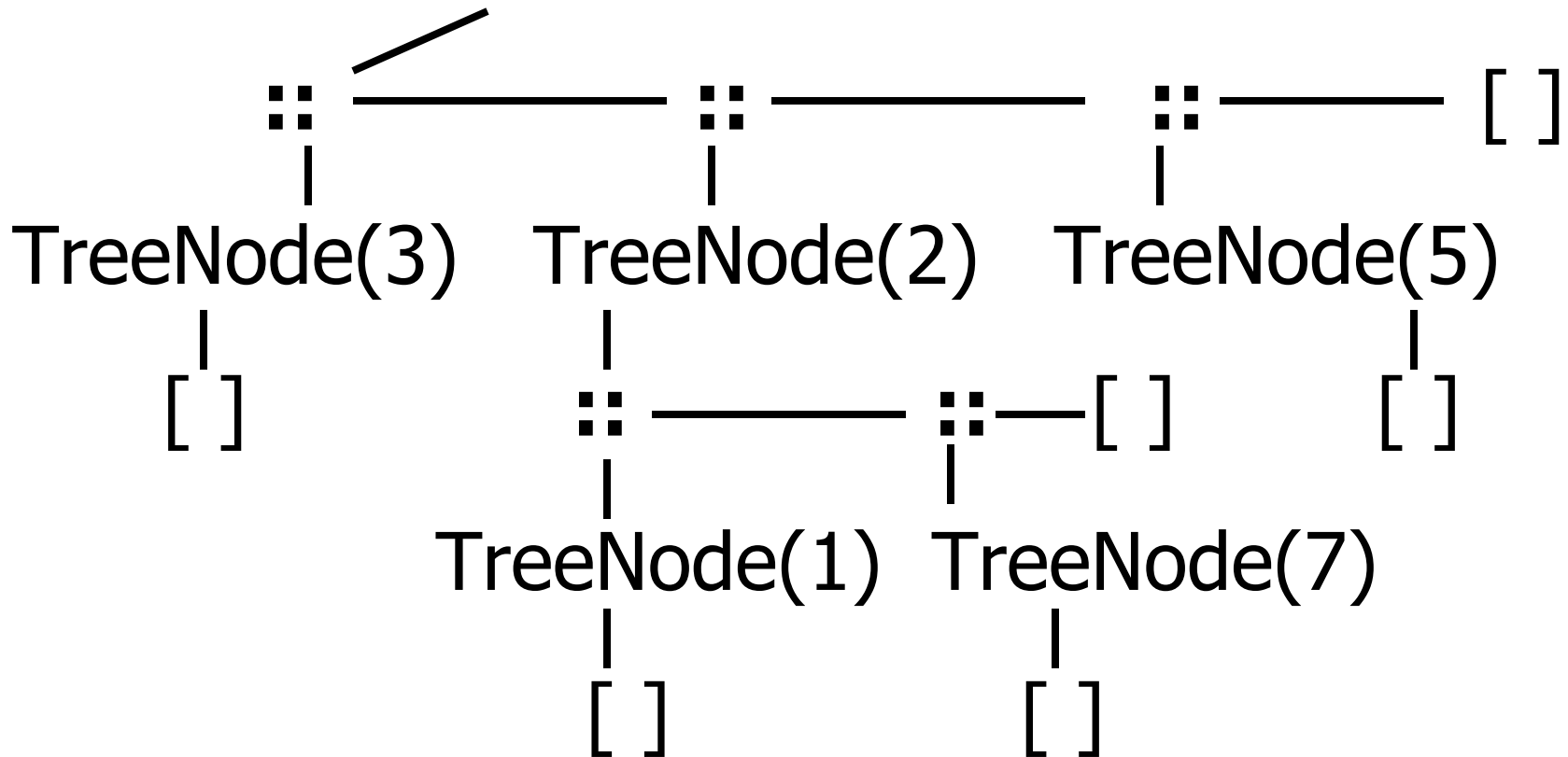


Nested Recursive Type Values

```
val ltree : int labeled_tree =  
  TreeNode  
    (5,  
     [TreeNode (3, []); TreeNode (2,  
      [TreeNode (1, []); TreeNode (7, [])]);  
     TreeNode (5, [])])
```

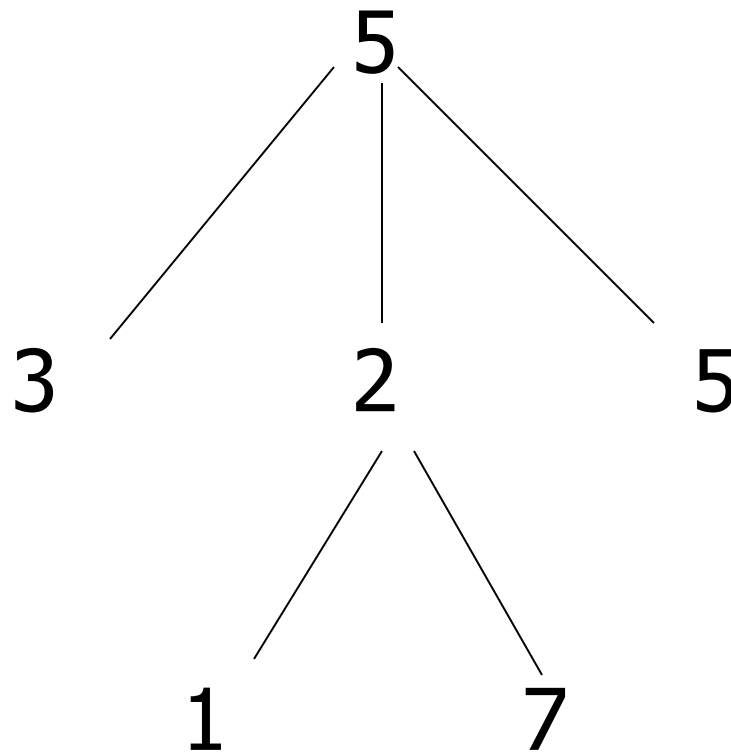
Nested Recursive Type Values

Ltree = TreeNode(5)





Nested Recursive Type Values





Mutually Recursive Functions

```
# let rec flatten_tree labtree =  
  match labtree with TreeNode (x,treelist)  
    -> x::flatten_tree_list treelist  
and flatten_tree_list treelist =  
  match treelist with [] -> []  
  | labtree::labtrees  
    -> flatten_tree labtree  
    @ flatten_tree_list labtrees;;
```



Mutually Recursive Functions

```
val flatten_tree : 'a labeled_tree -> 'a list =  
  <fun>
```

```
val flatten_tree_list : 'a labeled_tree list -> 'a  
  list = <fun>
```

```
# flatten_tree ltree;;
```

```
- : int list = [5; 3; 2; 1; 7; 5]
```

- Nested recursive types lead to mutually recursive functions



Why Data Types?

- Data types play a key role in:
 - *Data abstraction* in the design of programs
 - *Type checking* in the analysis of programs
 - *Compile-time code generation* in the translation and execution of programs
 - Data layout (how many words; which are data and which are pointers) dictated by type



Terminology

- Type: A **type** t defines a set of possible data values
 - E.g. **short** in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
 - A value in this set is said to have type t
- Type system: rules of a language assigning types to expressions



Types as Specifications

- Types describe properties
- Different type systems describe different properties, eg
 - Data is read-write versus read-only
 - Operation has authority to access data
 - Data came from “right” source
 - Operation might or could not raise an exception
- Common type systems focus on types describing same data layout and access methods



Sound Type System

- If an expression is assigned type t , and it evaluates to a value v , then v is in the set of values defined by t
- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not



Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is *strongly typed*
 - Eg: `1 + 2.3;;`
- Depends on definition of “type error”



Strongly Typed Language

- C++ claimed to be “strongly typed”, but
 - Union types allow creating a value at one type and using it at another
 - Type coercions may cause unexpected (undesirable) effects
 - No array bounds check (in fact, no runtime checks at all)
- SML, OCAML “strongly typed” but still must do dynamic array bounds checks, runtime type case analysis, and other checks



Static vs Dynamic Types

- *Static type*: type assigned to an expression at compile time
- *Dynamic type*: type assigned to a storage location at run time
- *Statically typed language*: static type assigned to every expression at compile time
- *Dynamically typed language*: type of an expression determined at run time



Type Checking

- When is $\text{op}(\text{arg1}, \dots, \text{argn})$ allowed?
- *Type checking* assures that operations are applied to the right number of arguments of the right types
 - Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- Used to resolve overloaded operations



Type Checking

- Type checking may be done *statically* at compile time or *dynamically* at run time
- Dynamically typed (aka untyped) languages (eg LISP, Prolog) do only dynamic type checking
- Statically typed languages can do most type checking statically



Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
 - Same variable may be used at different types



Dynamic Type Checking

- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe years after the code was written)



Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time



Static Type Checking

- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
 - Eg: array bounds



Static Type Checking

- Typically places restrictions on languages
 - Garbage collection
 - References instead of pointers
 - All variables initialized when created
 - Variable only used at one type
 - Union types allow for work-arounds, but effectively introduce dynamic type checks



Type Declarations

- ***Type declarations***: explicit assignment of types to variables (signatures to functions) in the code of a program
 - Must be checked in a strongly typed language
 - Often not necessary for strong typing or even static typing (depends on the type system)



Type Inference

- *Type inference*: A program analysis to assign a type to an expression from the program context of the expression
 - Fully static type inference first introduced by Robin Miller in ML
 - Haskell, OCAML, SML all use type inference
 - Records are a problem for type inference



Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \vdash \text{exp} : \tau$$

- Γ is a typing environment
 - Supplies the types of variables (and function names when function names are not variables)
 - Γ is a set of the form $\{ x : \sigma , \dots \}$
 - For any x at most one σ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- τ is a type to be assigned to exp
- \vdash pronounced “turnstile”, or “entails” (or “satisfies” or, informally, “shows”)



Axioms - Constants

$\Gamma \vdash n : \text{int}$ (assuming n is an integer constant)

$\Gamma \vdash \text{true} : \text{bool}$

$\Gamma \vdash \text{false} : \text{bool}$

- These rules are true with any typing environment
- Γ, n are meta-variables



Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$

Note: if such σ exists, its unique

Variable axiom:

$$\frac{}{\Gamma \vdash x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$



Simple Rules - Arithmetic

Primitive operators ($\oplus \in \{+, -, *, \dots\}$):

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad (\oplus) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3}{\Gamma \vdash e_1 \oplus e_2 : \tau_3}$$

Relations ($\sim \in \{<, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is **int**



Example: $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

What do we need to show first?

$\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$



Example: $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

What do we need for the left side?

$$\frac{\{x : \text{int}\} \vdash x + 2 : \text{int} \quad \{x:\text{int}\} \vdash 3 : \text{int}}{\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}} \text{Rel}$$



Example: $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

How to finish?

$$\frac{\frac{\{x:\text{int}\} \vdash x:\text{int} \quad \{x:\text{int}\} \vdash 2:\text{int}}{\{x:\text{int}\} \vdash x + 2 : \text{int}}^{\text{AO}} \quad \{x:\text{int}\} \vdash 3 : \text{int}}{\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}}^{\text{Rel}}$$



Example: $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

Complete Proof (type derivation)

$$\frac{\frac{\text{Var}}{\{x:\text{int}\} \vdash x:\text{int}} \quad \frac{\text{Const}}{\{x:\text{int}\} \vdash 2:\text{int}}}{\{x:\text{int}\} \vdash x + 2 : \text{int}} \text{AO} \quad \frac{\text{Const}}{\{x:\text{int}\} \vdash 3 : \text{int}}}{\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}} \text{Rel}$$



Simple Rules - Booleans

Connectives

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ || \ e_2 : \text{bool}}$$



Type Variables in Rules

- If_then_else rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- τ is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if_then_else must all have same type



Function Application

- Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 e_2) : \tau_2}$$

- If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$ applied to an argument e_2 of type τ_1 , the resulting expression $e_1 e_2$ has type τ_2



Fun Rule

- Rules describe types, but also how the environment Γ may change
- Can only do what rule allows!
- fun rule:

$$\frac{\{x : \tau_1\} + \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$



Fun Examples

$$\frac{\{y : \text{int}\} + \Gamma \vdash y + 3 : \text{int}}{\Gamma \vdash \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}}$$
$$\frac{\{f : \text{int} \rightarrow \text{bool}\} + \Gamma \vdash f \ 2 :: [\text{true}] : \text{bool list}}{\Gamma \vdash (\text{fun } f \rightarrow f \ 2 :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}}$$



(Monomorphic) Let and Let Rec

- let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

$$\frac{\{x : \tau_1\} + \Gamma \vdash e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$



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:
 - ② $\{one : int\ list\} \vdash$
 - ①

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

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



Proof of 1

- Which rule?

$\{\text{one} : \text{int list}\} \vdash (1 :: \text{one}) : \text{int list}$



Proof of 1

- Application

③

$$\frac{\{one : int\ list\} \vdash ((::) 1) : int\ list \rightarrow int\ list}{\{one : int\ list\} \vdash (1 :: one) : int\ list}$$

④

$$\frac{\{one : int\ list\} \vdash one : int\ list}{\{one : int\ list\} \vdash (1 :: one) : int\ list}$$



Proof of 3

Constants Rule

Constants Rule

$$\{one : int\ list\} \vdash$$

$$\{one : int\ list\} \vdash$$
$$(::) : int \rightarrow int\ list \rightarrow int\ list$$
$$1 : int$$

$$\{one : int\ list\} \vdash ((::) 1) : int\ list \rightarrow int\ list$$



Proof of 4

- Rule for variables

$$\frac{}{\{one : int\ list\} \vdash one:int\ list}$$



Proof of 2

- Constant

⑤ $\{x:\text{int}; \text{one} : \text{int list}\} \vdash$
 $\text{fun } y \rightarrow$
 $(x :: y :: \text{one}))$

$\{\text{one} : \text{int list}\} \vdash 2:\text{int} \quad : \text{int} \rightarrow \text{int list}$

$\{\text{one} : \text{int list}\} \vdash (\text{let } x = 2 \text{ in}$
 $\text{fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$



Proof of 5

?

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



Proof of 5

?

$$\{y:\text{int}; x:\text{int}; \text{one} : \text{int list}\} \vdash (x :: y :: \text{one}) : \text{int list}$$

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



Proof of 5

⑥

$$\frac{\{y:\text{int}; x:\text{int}; \text{one}:\text{int list}\} \quad \{y:\text{int}; x:\text{int}; \text{one}:\text{int list}\} \\ \vdash ((::) x):\text{int list} \rightarrow \text{int list}}{\{y:\text{int}; x:\text{int}; \text{one} : \text{int list}\} \vdash (x :: y :: \text{one}) : \text{int list}}$$

⑦

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



Proof of 6

Constant

Variable

$\{\dots\} \vdash (::)$

$: \text{int} \rightarrow \text{int list} \rightarrow \text{int list}$

$\{\dots; x:\text{int}; \dots\} \vdash x:\text{int}$

$\{y:\text{int}; x:\text{int}; \text{one} : \text{int list}\} \vdash ((::) x)$

$: \text{int list} \rightarrow \text{int list}$



Proof of 7

Pf of 6 [y/x]

Variable

•
•
•

$$\frac{\frac{\{y:\text{int}; \dots\} \vdash ((::) y) : \text{int list} \rightarrow \text{int list}}{\{y:\text{int}; x:\text{int}; \text{one} : \text{int list}\} \vdash (y :: \text{one}) : \text{int list}} \quad \frac{\{\dots; \text{one} : \text{int list}\} \vdash \text{one} : \text{int list}}{\{y:\text{int}; x:\text{int}; \text{one} : \text{int list}\} \vdash (y :: \text{one}) : \text{int list}}$$



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

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



Mea Culpa

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



Support for Polymorphic Types

- Monomorphic 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
 - $\forall \alpha_1, \dots, \alpha_n . \tau$
 - 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 $\text{FreeVars}(\tau)$
- Free variables of polymorphic type removes variables that are universally quantified
 - $\text{FreeVars}(\forall \alpha_1, \dots, \alpha_n . \tau) = \text{FreeVars}(\tau) - \{\alpha_1, \dots, \alpha_n\}$
- $\text{FreeVars}(\Gamma) =$ all FreeVars of types in range of Γ



Monomorphic to Polymorphic

- Given:
 - type environment Γ
 - monomorphic type τ
 - τ shares type variables with Γ
- Want most polymorphic type for τ that doesn't break sharing type variables with Γ
- $\text{Gen}(\tau, \Gamma) = \forall \alpha_1, \dots, \alpha_n . \tau$ where
 $\{\alpha_1, \dots, \alpha_n\} = \text{freeVars}(\tau) - \text{freeVars}(\Gamma)$



Polymorphic Typing Rules

- A *type judgement* has the form
$$\Gamma \vdash \text{exp} : \tau$$
 - Γ uses **polymorphic** types
 - τ still monomorphic
- Most rules stay same (except use more general typing environments)
- Rules that change:
 - Variables
 - Let and Let Rec
 - Allow polymorphic constants
- Worth noting functions again



Polymorphic Let and Let Rec

■ let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \{x : \text{Gen}(\tau_1, \Gamma)\} + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

■ let rec rule:

$$\frac{\{x : \tau_1\} + \Gamma \vdash e_1 : \tau_1 \quad \{x : \text{Gen}(\tau_1, \Gamma)\} + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$

Polymorphic Variables (Identifiers)

Variable axiom:

$$\overline{\Gamma \vdash x : \varphi(\tau)} \quad \text{if } \Gamma(x) = \forall \alpha_1, \dots, \alpha_n . \tau$$

- Where φ replaces all occurrences of $\alpha_1, \dots, \alpha_n$ by monotypes τ_1, \dots, τ_n
- Note: Monomorphic rule special case:

$$\overline{\Gamma \vdash x : \tau} \quad \text{if } \Gamma(x) = \tau$$

- Constants treated same way



Fun Rule Stays the Same

- fun rule:

$$\frac{\{x : \tau_1\} + \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

- Types τ_1, τ_2 monomorphic
- Function argument must always be used at same type in function body



Polymorphic Example

- Assume additional constants:
- $\text{hd} : \forall \alpha. \alpha \text{ list} \rightarrow \alpha$
- $\text{tl} : \forall \alpha. \alpha \text{ list} \rightarrow \alpha \text{ list}$
- $\text{is_empty} : \forall \alpha. \alpha \text{ list} \rightarrow \text{bool}$
- $:: : \forall \alpha. \alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$
- $[] : \forall \alpha. \alpha \text{ list}$



Polymorphic Example

- Show:

?

```
{ } |- let rec length =  
    fun l -> if is_empty l then 0  
             else 1 + length (tl l)  
in length ((::) 2 []) + length((::) true []) : int
```

Polymorphic Example: Let Rec Rule

■ Show: (1) (2)

$\{\text{length}:\alpha \text{ list} \rightarrow \text{int}\}$	$\{\text{length}:\forall\alpha. \alpha \text{ list} \rightarrow \text{int}\}$
$\vdash \text{fun } l \rightarrow \dots$	$\vdash \text{length } ((::) 2 []) +$
$:\alpha \text{ list} \rightarrow \text{int}$	$\text{length}((::) \text{true } []) : \text{int}$

$\{\}$ $\vdash \text{let rec length} =$

$\text{fun } l \rightarrow \text{if is_empty } l \text{ then } 0$
 $\quad \text{else } 1 + \text{length } (\text{tl } l)$

$\text{in length } ((::) 2 []) + \text{length}((::) \text{true } []) : \text{int}$



Polymorphic Example (1)

- Show:

?

```
{length:  $\alpha$  list -> int} |-  
fun l -> if is_empty l then 0  
        else 1 + length (tl l)  
:  
 $\alpha$  list -> int
```




Polymorphic Example (1): Fun Rule

■ Show: (3)

$\{ \text{length} : \alpha \text{ list} \rightarrow \text{int}, \text{ l} : \alpha \text{ list} \} \vdash$

if is_empty l then 0

else length (hd l) + length (tl l) : int

$\{ \text{length} : \alpha \text{ list} \rightarrow \text{int} \} \vdash$

fun l -> if is_empty l then 0

else 1 + length (tl l)

: $\alpha \text{ list} \rightarrow \text{int}$



Polymorphic Example (3)

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, \text{ l} : \alpha \text{ list} \}$
- Show

?

$\Gamma \vdash \text{if is_empty l then 0}$
 $\quad \text{else } 1 + \text{length (tl l)} : \text{int}$



Polymorphic Example (3): IfThenElse

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, l : \alpha \text{ list}\}$
- Show

$$\begin{array}{ccc} (4) & (5) & (6) \\ \Gamma \vdash \text{is_empty } l & \Gamma \vdash 0 : \text{int} & \Gamma \vdash 1 + \\ & : \text{bool} & \text{length } (\text{tl } l) : \text{int} \\ \hline \Gamma \vdash \text{if is_empty } l \text{ then } 0 \\ & \text{else } 1 + \text{length } (\text{tl } l) : \text{int} \end{array}$$



Polymorphic Example (4)

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, \text{ l} : \alpha \text{ list} \}$
- Show

?

$\Gamma \vdash \text{is_empty l} : \text{bool}$



Polymorphic Example (4): Application

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, \text{ l} : \alpha \text{ list}\}$
- Show

?

?

$$\Gamma \vdash \text{is_empty} : \alpha \text{ list} \rightarrow \text{bool}$$

$$\Gamma \vdash \text{l} : \alpha \text{ list}$$

$$\Gamma \vdash \text{is_empty l} : \text{bool}$$



Polymorphic Example (4)

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, \text{ l} : \alpha \text{ list}\}$
- Show

By Const since $\alpha \text{ list} \rightarrow \text{bool}$ is
instance of $\forall \alpha. \alpha \text{ list} \rightarrow \text{bool}$?

$$\Gamma \vdash \text{is_empty} : \alpha \text{ list} \rightarrow \text{bool}$$

$$\Gamma \vdash \text{l} : \alpha \text{ list}$$

$$\Gamma \vdash \text{is_empty l} : \text{bool}$$



Polymorphic Example (4)

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, l : \alpha \text{ list}\}$
- Show

By Const since $\alpha \text{ list} \rightarrow \text{bool}$ is instance of $\forall \alpha. \alpha \text{ list} \rightarrow \text{bool}$ By Variable $\Gamma(l) = \alpha \text{ list}$

$$\Gamma \vdash \text{is_empty} : \alpha \text{ list} \rightarrow \text{bool}$$

$$\Gamma \vdash l : \alpha \text{ list}$$

$$\Gamma \vdash \text{is_empty } l : \text{bool}$$

- This finishes (4)



Polymorphic Example (5):Const

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, \text{ l} : \alpha \text{ list} \}$

- Show

By Const Rule

$$\frac{}{\Gamma \vdash 0 : \text{int}}$$

Polymorphic Example (6): Arith Op

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, l : \alpha \text{ list}\}$
- Show

By Variable

$$\frac{}{\Gamma \vdash \text{length}} \quad (7)$$

By Const

$$\frac{}{1 : \alpha \text{ list} \rightarrow \text{int}} \quad \Gamma \vdash (tl\ l) : \alpha \text{ list}$$

$$\frac{}{\Gamma \vdash 1 : \text{int}}$$

$$\frac{}{\Gamma \vdash \text{length} (tl\ l) : \text{int}}$$

$$\frac{}{\Gamma \vdash 1 + \text{length} (tl\ l) : \text{int}}$$



Polymorphic Example (7):App Rule

- Let $\Gamma = \{\text{length} : \alpha \text{ list} \rightarrow \text{int}, l : \alpha \text{ list}\}$
- Show

By Const	By Variable
$\frac{}{\Gamma \vdash (\text{tl } l) : \alpha \text{ list} \rightarrow \alpha \text{ list}}$	$\frac{}{\Gamma \vdash l : \alpha \text{ list}}$
$\Gamma \vdash (\text{tl } l) : \alpha \text{ list}$	

By Const since $\alpha \text{ list} \rightarrow \alpha \text{ list}$ is instance of
 $\forall \alpha. \alpha \text{ list} \rightarrow \alpha \text{ list}$

Polymorphic Example: (2) by ArithOp

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:

(8)

$\Gamma' \vdash$

$\text{length} ((::) 2 []) : \text{int}$

(9)

$\Gamma' \vdash$

$\text{length} ((::) \text{true} []) : \text{int}$

$\{\text{length}: \alpha. \alpha \text{ list} \rightarrow \text{int}\}$

$\vdash \text{length} ((::) 2 []) + \text{length} ((::) \text{true} []) : \text{int}$



Polymorphic Example: (8)AppRule

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:

$$\frac{\Gamma' \vdash \text{length} : \text{int list} \rightarrow \text{int} \quad \Gamma' \vdash ((::)2 []): \text{int list}}{\text{list}}$$
$$\Gamma' \vdash \text{length } ((::) 2 []) : \text{int}$$



Polymorphic Example: (8)AppRule

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$

- Show:

By Var since $\text{int list} \rightarrow \text{int}$ is instance of

$\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$

(10)

$$\frac{\Gamma' \vdash \text{length} : \text{int list} \rightarrow \text{int} \quad \Gamma' \vdash ((::)2 []):\text{int}}{\text{list}}$$
$$\Gamma' \vdash \text{length } ((::)2 []) : \text{int}$$



Polymorphic Example: (10)AppRule

- Let $\Gamma' = \{\text{length}.\forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
- By Const since $\alpha \text{ list}$ is instance of $\forall \alpha. \alpha \text{ list}$

(11)

$$\frac{\Gamma' \vdash ((::) 2) : \text{int list} \rightarrow \text{int list} \quad \overline{\Gamma' \vdash [] : \text{int list}}}{\Gamma' \vdash ((::) 2 []) : \text{int list}}$$

Polymorphic Example: (11)AppRule

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
- By Const since $\alpha \text{ list}$ is instance of

$$\frac{\forall \alpha. \alpha \text{ list}}{\Gamma' \vdash ((::) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list}) : \text{int list} \rightarrow \text{int list}} \quad \frac{\text{By Const}}{\Gamma' \vdash 2 : \text{int}}$$

$$\Gamma' \vdash ((::) 2) : \text{int list} \rightarrow \text{int list}$$



Polymorphic Example: (9)AppRule

- Let $\Gamma' = \{\text{length}.\forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:

$$\frac{\Gamma' \vdash \text{length}:\text{bool list} \rightarrow \text{int} \quad \Gamma' \vdash ((::) \text{true } []):\text{bool list}}{\Gamma' \vdash \text{length } ((::) \text{true } []) : \text{int}}$$



Polymorphic Example: (9)AppRule

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$

- Show:

By Var since $\text{bool list} \rightarrow \text{int}$ is instance of

$\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$

(12)

 $\Gamma' \vdash$ $\text{length} : \text{bool list} \rightarrow \text{int}$ $\Gamma' \vdash$ $((::) \text{ true } []): \text{bool list}$

 $\Gamma' \vdash \text{length } ((::) \text{ true } []) : \text{int}$



Polymorphic Example: (12)AppRule

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
- By Const since $\alpha \text{ list}$ is instance of $\forall \alpha. \alpha \text{ list}$

(13)

$$\frac{\Gamma' \vdash ((::)\text{true}): \text{bool list} \rightarrow \text{bool list} \quad \overline{\Gamma' \vdash []: \text{bool list}}}{\text{list}}$$
$$\Gamma' \vdash ((::)\text{true} []) : \text{bool list}$$

Polymorphic Example: (13)AppRule

- Let $\Gamma' = \{\text{length} \forall \alpha. \alpha \text{ list} \rightarrow \text{int}\}$

- Show:

By Const since bool list

is instance of $\forall \alpha. \alpha \text{ list}$

By Const

$$\begin{array}{c}
 \hline
 \Gamma' \vdash \\
 \hline
 ((::) : \text{bool} \rightarrow \text{bool list} \rightarrow \text{bool list})
 \end{array}
 \quad
 \begin{array}{c}
 \hline
 \Gamma' \vdash \\
 \hline
 \text{true} : \text{bool}
 \end{array}$$

$$\Gamma' \vdash ((::) \text{ true}) : \text{bool list} \rightarrow \text{bool list}$$