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)
```

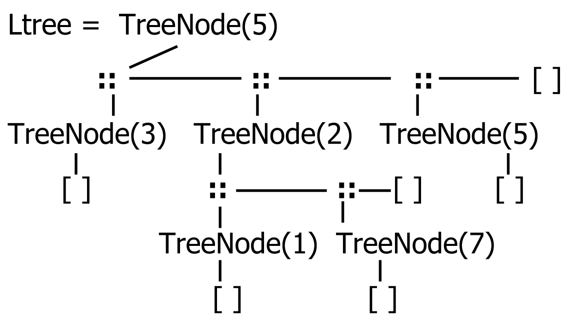

Nested Recursive Type Values

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

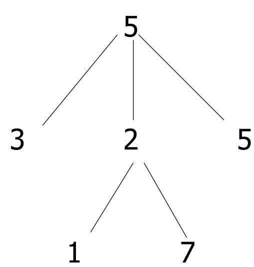

Nested Recursive Type Values

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


Nested Recursive Type Values



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 op(arg1,…,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
  - Haskle, OCAML, SML all use type inference
    - Records are a problem for type inference

Format of Type Judgments

- A *type judgement* has the form

  Γ |- exp : τ
- 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 :σ , . . . }
  - 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
- |- pronounced "turnstyle", or "entails" (or "satisfies" or, informally, "shows")

Axioms - Constants

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

$$\frac{}{\Gamma \text{ |- true : bool}} \qquad \frac{}{\Gamma \text{ |- false : bool}}$$

- 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 exits, its unique

Variable axiom:

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

Simple Rules - Arithmetic

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

$$\frac{\Gamma \text{ |- } e_1{:}\tau_1 \qquad \Gamma \text{ |- } e_2{:}\tau_2 \qquad (\oplus){:}\tau_1 \rightarrow \tau_2 \rightarrow \tau_3}{\Gamma \text{ |- } e_1 \oplus e_2 : \tau_3}$$

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

$$\frac{\Gamma \text{ |- } e_1 : \tau \qquad \Gamma \text{ |- } e_2 : \tau}{\Gamma \text{ |- } e_1 \sim e_2 \text{ :bool}}$$

For the moment, think τ is int

Example: {x:int} |- x + 2 = 3 :bool

What do we need to show first?

{x:int} |- x + 2 = 3 : bool

Example: {x:int} |- x + 2 = 3 :bool

What do we need for the left side?

$$\frac{\text{\{x : int\} |- x + 2 : int} \qquad \text{\{x:int\} |- 3 :int}}{\text{\{x:int\} |- x + 2 = 3 : bool}}\text{Rel}$$

Example: {x:int} |- x + 2 = 3 :bool

How to finish?

$$\dfrac{\dfrac{\{x{:}int\} \mid\!- x{:}int \quad \{x{:}int\} \mid\!- 2{:}int}{\{x : int\} \mid\!- x + 2 : int}\text{AO} \qquad \{x{:}int\} \mid\!- 3 : int}{\{x{:}int\} \mid\!- x + 2 = 3 : bool}\text{Rel}$$

Example: {x:int} |- x + 2 = 3 :bool

Complete Proof (type derivation)

$$\dfrac{\dfrac{\dfrac{}{\{x{:}int\} \mid\!- x{:}int}\text{Var} \quad \dfrac{}{\{x{:}int\} \mid\!- 2{:}int}\text{Const}}{\{x : int\} \mid\!- x + 2 : int}\text{AO} \qquad \dfrac{}{\{x{:}int\} \mid\!- 3 : int}\text{Const}}{\{x{:}int\} \mid\!- x + 2 = 3 : bool}\text{Rel}$$

Simple Rules - Booleans

Connectives

$$\dfrac{\Gamma \mid\!- e_1 : bool \quad \Gamma \mid\!- e_2 : bool}{\Gamma \mid\!- e_1 \;\&\&\; e_2 : bool}$$

$$\dfrac{\Gamma \mid\!- e_1 : bool \quad \Gamma \mid\!- e_2 : bool}{\Gamma \mid\!- e_1 \;||\; e_2 : bool}$$

Type Variables in Rules

- If_then_else rule:

$$\dfrac{\Gamma \mid\!- e_1 : bool \quad \Gamma \mid\!- e_2 : \tau \quad \Gamma \mid\!- e_3 : \tau}{\Gamma \mid\!- (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- 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:

$$\dfrac{\Gamma \mid\!- e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \mid\!- e_2 : \tau_1}{\Gamma \mid\!- (e_1\ e_2) : \tau_2}$$

- 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:

$$\dfrac{\{x : \tau_1\} + \Gamma \mid\!- e : \tau_2}{\Gamma \mid\!- \text{fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

Fun Examples

$$\frac{\{y : int\} + \Gamma \mid\!- \; y + 3 : int}{\Gamma \mid\!- \; \text{fun } y \to y + 3 : int \to int}$$

$$\frac{\{f : int \to bool\} + \Gamma \mid\!- \; f\ 2 :: [true] : bool\ list}{\Gamma \mid\!- \; (\text{fun } f \to f\ 2 :: [true]) : (int \to bool) \to bool\ list}$$

(Monomorphic) Let and Let Rec

- let rule:

$$\frac{\Gamma \mid\!- \; e_1 : \tau_1 \qquad \{x : \tau_1\} + \Gamma \mid\!- \; e_2 : \tau_2}{\Gamma \mid\!- \; (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

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

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:

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

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

(conclusion below the line, from premises (1) and (2)):

|- (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

- Application

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

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

(conclusion below the line, from premises (3) and (4)):

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

Proof of 3

Constants Rule　　　　　Constants Rule

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

---

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

Proof of 4

- Rule for variables

{one : int list} |- one:int list

Proof of 2

- Constant

{one : int list} |- 2:int　　⑤ {x:int; one : int list} |- fun y -> (x :: y :: one)) : int → int list

---

{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

Proof of 5

⑥ {y:int; x:int; one:int list} |- ((::) x):int list→ int list　　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 6

Constant   Variable

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

Proof of 7

Pf of 6 [y/x]   Variable

⋮

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

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

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

- 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

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
- Gen(τ, Γ) = $\forall\alpha_1, \ldots, \alpha_n . \tau$ where $\{\alpha_1, \ldots, \alpha_n\}$ = freeVars(τ) – freeVars(Γ)

Polymorphic Typing Rules

- A *type judgement* has the form
  Γ |- exp : τ
  - 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:

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

- let rec rule:

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

Polymorphic Variables (Identifiers)

Variable axiom:

$$\frac{}{\Gamma \mid\!- x : \varphi(\tau)} \quad \text{if } \Gamma(x) = \forall\alpha_1, \ldots, \alpha_n . \tau$$

- Where φ replaces all occurrences of $\alpha_1, \ldots, \alpha_n$ by monotypes $\tau_1, \ldots, \tau_n$
- Note: Monomorphic rule special case:

$$\frac{}{\Gamma \mid\!- x : \tau} \quad \text{if } \Gamma(x) = \tau$$

- Constants treated same way

Fun Rule Stays the Same

- fun rule:

$$\frac{\{x : \tau_1\} + \Gamma \mid\!- e : \tau_2}{\Gamma \mid\!- \text{fun } x \text{ -> } e : \tau_1 \to \tau_2}$$

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

Polymorphic Example

- Assume additional constants:
- hd : $\forall\alpha.\ \alpha$ list -> α
- tl: $\forall\alpha.\ \alpha$ list -> α list
- is_empty : $\forall\alpha.\ \alpha$ list -> bool
- :: : $\forall\alpha.\ \alpha$ -> α list -> α list
- [] : $\forall\alpha.\ \alpha$ 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)

(1) {length:α list -> int} |- fun l -> … : α list -> int

(2) {length:$\forall\alpha.\ \alpha$ list -> int} |- length ((::) 2 []) + length((::) true []) : int

---

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

Polymorphic Example (1)

- Show:

?

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

Polymorphic Example (1): Fun Rule

- Show: (3)

{length:α list -> int, l: α list } |-
if is_empty l then 0
else length (hd l) + length (tl l) : int

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

Polymorphic Example (3)

- Let Γ ={length:α list -> int, l: α list }
- Show

?

Γ|- if is_empty l then 0
else 1 + length (tl l) : int

Polymorphic Example (3):IfThenElse

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\overset{(4)}{\Gamma |\text{- is\_empty l : bool}} \quad \overset{(5)}{\Gamma |\text{- 0:int}} \quad \overset{(6)}{\Gamma |\text{- 1 + length (tl l) : int}}}{\Gamma |\text{- if is\_empty l then 0 else 1 + length (tl l) : int}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{?}{\Gamma |\text{- is\_empty l : bool}}$$

Polymorphic Example (4):Application

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{?}{\Gamma |\text{- is\_empty : } \alpha \text{ list -> bool}} \quad \dfrac{?}{\Gamma |\text{- l : } \alpha \text{ list}}}{\Gamma |\text{- is\_empty l : bool}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{\text{By Const since } \alpha \text{ list -> bool is instance of } \forall\alpha.\ \alpha \text{ list -> bool}}{\Gamma |\text{- is\_empty : } \alpha \text{ list -> bool}} \quad \dfrac{?}{\Gamma |\text{- l : } \alpha \text{ list}}}{\Gamma |\text{- is\_empty l : bool}}$$

Polymorphic Example (4)

- Let Γ ={length:α list -> int, l: α list }
- Show

$$\frac{\dfrac{\text{By Const since } \alpha \text{ list -> bool is instance of } \forall\alpha.\ \alpha \text{ list -> bool}}{\Gamma |\text{- is\_empty : } \alpha \text{ list -> bool}} \quad \dfrac{\text{By Variable } \Gamma(\text{l}) = \alpha \text{ list}}{\Gamma |\text{- l : } \alpha \text{ list}}}{\Gamma |\text{- is\_empty l : bool}}$$

- This finishes (4)

Polymorphic Example (5):Const

- Let Γ ={length:α list -> int, l: α list }
- Show

By Const Rule

$$\frac{}{\Gamma |\text{- 0:int}}$$

Polymorphic Example (6):Arith Op

- Let Γ ={length:α list -> int, l: α list }
- Show

By Variable
Γ|- length : α list -> int (7) Γ|- (tl l) : α list
Γ|- length (tl l) : int

By Const
Γ|- 1:int

Γ|- 1 + length (tl l) : int

Polymorphic Example (7):App Rule

- Let Γ ={length:α list -> int, l: α list }
- Show

By Const
Γ|- (tl l) : α list -> α list

By Variable
Γ|- l : α list

Γ|- (tl l) : α list

By Const since α list -> α list is instance of
$\forall\alpha$. α list -> α list

Polymorphic Example: (2) by ArithOp

- Let Γ' = {length:$\forall$ α. α list -> int}
- Show:

(8) Γ' |- length ((::) 2 []) :int

(9) Γ' |- length((::) true []) : int

{length: α. α list -> int}
|- length ((::) 2 []) + length((::) true []) : int

Polymorphic Example: (8)AppRule

- Let Γ' = {length:$\forall$ α. α list -> int}
- Show:

Γ' |- length : int list ->int Γ' |- ((::)2 []):int list

Γ' |- length ((::) 2 []) :int

Polymorphic Example: (8)AppRule

- Let Γ' = {length:$\forall$ α. α list -> int}
- Show:

By Var since int list -> int is instance of
$\forall\alpha$. α list -> int

(10)

Γ' |- length : int list ->int Γ' |- ((::)2 []):int list

Γ' |- length ((::) 2 []) :int

Polymorphic Example: (10)AppRule

- Let Γ' = {length:$\forall$ α. α list -> int}
- Show:
- By Const since α list is instance of
  $\forall\alpha$. α list

(11)

Γ' |-((::) 2) : int list -> int list Γ' |- [] : int list

Γ' |- ((::) 2 []) :int list

Polymorphic Example: (11)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:
- By Const since α list is instance of ∀α. α list

By Const

Γ' |- (::) : int -> int list -> int list   Γ' |- 2 : int

Γ' |- ((::) 2) : int list -> int list

Polymorphic Example: (9)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:

Γ' |- length:bool list ->int   Γ' |- ((::) true []):bool list

Γ' |- length ((::) true []) :int

Polymorphic Example: (9)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:

By Var since bool list -> int is instance of ∀α. α list -> int

(12)

Γ' |- length:bool list ->int   Γ' |- ((::) true []):bool list

Γ' |- length ((::) true []) :int

Polymorphic Example: (12)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:
- By Const since α list is instance of ∀α. α list

(13)

Γ' |-((::)true):bool list ->bool list   Γ' |- []:bool list

Γ' |- ((::) true []) :bool list

Polymorphic Example: (13)AppRule

- Let Γ' = {length∀ α. α list -> int}
- Show:

By Const since bool list is instance of ∀α. α list

By Const

Γ' |- (::):bool ->bool list ->bool list   Γ' |- true : bool

Γ' |- ((::) true) : bool list -> bool list