Nested Recursive Types

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

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

Ltree =  TreeNode(5) 
          ::                ::                 ::       ... 

                      TreeNode(1)  TreeNode(7) 

                       [ ]              [ ]
Mutually Recursive Functions

```ml
# 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;;
```

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 \( \{ x | 2^{15} - 1 \geq x \geq -2^{15} \} \)
  - 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”

**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

---

**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
  \[ \Gamma |- \text{exp} : \tau \]
- \( \Gamma \) is a typing environment
  - Supplies the types of variables (and function names when function names are not variables)
  - \( \Gamma \) is a set of the form \( \{x : \sigma, \ldots\} \)
  - For any \( x \) at most one \( \sigma \) such that \( (x : \sigma \in \Gamma) \)
- \( \text{exp} \) is a program expression
- \( \tau \) is a type to be assigned to \( \text{exp} \)
- \( |- \) pronounced “turnstyle”, or “entails” (or “satisfies” or, informally, “shows”)

Axioms - Constants

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

\[ \Gamma |- \text{true} : \text{bool} \quad \Gamma |- \text{false} : \text{bool} \]

- These rules are true with any typing environment
- \( \Gamma, n \) are meta-variables

Axioms – Variables (Monomorphic Rule)

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

Note: if such \( \sigma \) exits, its unique

Variable axiom:

\[ \Gamma |- x : \sigma \quad \text{if} \quad \Gamma(x) = \sigma \]

Simple Rules - Arithmetic

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

\[ \begin{align*}
\Gamma |- e_1 : \tau_1 & \quad \Gamma |- e_2 : \tau_2 \quad (\oplus): \tau_1 \to \tau_2 \to \tau_3 \\
\Gamma |- e_1 \oplus e_2 : \tau_3
\end{align*} \]

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

\[ \begin{align*}
\Gamma |- e_1 : \tau & \quad \Gamma |- e_2 : \tau \\
\Gamma |- e_1 \sim e_2 : \text{bool}
\end{align*} \]

For the moment, think \( \tau \) is int

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

What do we need to show first?

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

What do we need for the left side?

\{x : int\} |- x + 2 : int \quad \{x : int\} |- 3 : int

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

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

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

10/3/17

How to finish?

Complete Proof (type derivation)

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

10/3/17

Simple Rules - Booleans

Connectives

Γ |- e1 : bool  Γ |- e2 : bool
Γ |- e1 && e2 : bool

Γ |- e1 : bool  Γ |- e2 : bool
Γ |- e1 || e2 : bool

10/3/17

Type Variables in Rules

- If_then_else rule:

Γ |- e1 : bool  Γ |- e2 : τ  Γ |- e3 : τ
Γ |- (if e1 then e2 else e3) : τ

- 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

10/3/17

Function Application

- Application rule:

Γ |- e1 : τ1 → τ2  Γ |- e2 : τ1
Γ |- (e1 e2) : τ2

- If you have a function expression e1 of type τ1 → τ2 applied to an argument e2 of type τ1, the resulting expression e1 e2 has type τ2

10/3/17

Fun Rule

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

{x : τ1} + Γ |- e : τ2
Γ |- fun x -> e : τ1 → τ2
Fun Examples

\{y : \text{int}\} + \Gamma |- y + 3 : \text{int} \\
\Gamma |- \text{fun } y -> y + 3 : \text{int} \rightarrow \text{int} \\
\{f : \text{int} \rightarrow \text{bool}\} + \Gamma |- f \ 2 :: [\text{true}] : \text{bool\ list} \\
\Gamma |- (\text{fun } f -> f \ 2 :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool\ list}

(Monomorphic) Let and Let Rec

\text{let rule:}
\[
\Gamma |- e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma |- e_2 : \tau_2 \\
\Gamma |- (\text{let } x = e_1 \text{ in } e_2) : \tau_2
\]

\text{let rec rule:}
\[
\{x : \tau_1\} + \Gamma |- e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma |- e_2 : \tau_2 \\
\Gamma |- (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2
\]

Example

\text{Which rule do we apply?}

\[
\text{let rec \ one = 1 :: \ one in} \\
\text{let \ x = 2 in} \\
\text{fun \ y -> (x :: y :: \ one)} : \text{int} \rightarrow \text{int\ list}
\]

Example

\text{Let rec rule:}
\[
\begin{align*}
\{\text{one : int list}\} & |- \\
\{\text{one : int list}\} & |- \text{fun } y -> (x :: y :: \ one) \\
\{\text{one : int list}\} & |- (1 :: \ one) : \text{int list} \\
\{\text{one : int list}\} & |- (\text{let rec } \ one = 1 :: \ one \text{ in} \ \\
\{\text{one : int list}\} & |- \text{let } x = 2 \text{ in} \\
\{\text{one : int list}\} & |- \text{fun } y -> (x :: y :: \ one) : \text{int} \rightarrow \text{int\ list}
\end{align*}
\]

Proof of 1

\text{Which rule?}

\[
\{\text{one : int list}\} |- (1 :: \ one) : \text{int list}
\]

Proof of 1

\text{Application}
\[
\begin{align*}
\{\text{one : int list}\} & |- \\
\{\text{one : int list}\} & |- ((::) 1) : \text{int list} \rightarrow \text{int list} \\
\{\text{one : int list}\} & |- (1 :: \ one) : \text{int list}
\end{align*}
\]
Proof of 3

Constants Rule               Constants Rule
{one : int list} |-                  {one : int list} |- 
(::) : int → int list→ 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

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

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

Proof of 5

6                                        7
{y:int; x:int; one : int list}     {y:int; x:int; one:int list}
|- ((::) x):int list→ int list  |- (y :: one) : int list 
|- ((::) 1) : int list→ 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 5

6
7
{y:int; x:int; one:int list}     {y:int; x:int; one:int list}
|- ((::) x):int list→ int list  |- (y :: one) : int list 
|- ((::) 1) : int list→ 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 \begin{align*} \{ \ldots \} & \vdash (\vdash) \end{align*}  
Variable  
: \text{int} \to \text{int list} \to \text{int list}  
\{y: \text{int}; x: \text{int}; \text{one} : \text{int list}\} & \vdash (\vdash) x \end{align*}  
: \text{int list} \to \text{int list}

Proof of 7

Pf of 6 \[y/x\]  
Variable  
\begin{align*} \{y: \text{int}; \ldots\} & \vdash ((\vdash) y) \\ \{\ldots; \text{one}: \text{int list}\} & \vdash (\vdash) x \\ \text{int list} \to \text{int list} & \vdash (y :: \text{one}) : \text{int list} \end{align*}

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  
\[
\frac{A \Rightarrow B}{A} \quad \frac{B}{B}
\]

- 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  
  - \texttt{let} and \texttt{let rec} rules to introduce polymorphism  
  - Explicit rule to eliminate (instantiate) polymorphism

Support for Polymorphic Types

- Monomorphic Types ($\tau$):  
  - Basic Types: \texttt{int}, \texttt{bool}, \texttt{float}, \texttt{string}, \texttt{unit}, \ldots  
  - Type Variables: $\alpha$, $\beta$, $\gamma$, $\delta$, $\varepsilon$  
  - Compound Types: $\alpha \Rightarrow \beta$, \texttt{int} * \texttt{string}, \texttt{bool} list, \ldots

- Polymorphic Types:  
  - Monomorphic types $\tau$  
  - Universally quantified monomorphic types  
    \[
    \forall \alpha_1, \ldots, \alpha_n. \tau
    \]
  - Can think of $\tau$ as same as $\forall. \tau$
Support for Polymorphic Types

- Typing Environment $\Gamma$ 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
  - $\text{FreeVars}(\alpha_1, \ldots, \alpha_n . \tau) = \text{FreeVars}(\tau) - \{\alpha_1, \ldots, \alpha_n\}$
- $\text{FreeVars}(\Gamma) =$ all FreeVars of types in range of $\Gamma$

Monomorphic to Polymorphic

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

Polymorphic Typing Rules

- A type judgement has the form $\Gamma |- \text{exp} : \tau$
  - $\Gamma$ uses polymorphic types
  - $\tau$ 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 Variables (Identifiers)

- Variable axiom:
  - $\Gamma |- x : \phi(\tau)$ if $\Gamma(x) = \Psi\alpha_1, \ldots, \alpha_n . \tau$
  - Where $\phi$ replaces all occurrences of $\alpha_1, \ldots, \alpha_n$ by monotypes $\tau_1, \ldots, \tau_n$
- Note: Monomorphic rule special case:
  - $\Gamma |- x : \tau$ if $\Gamma(x) = \tau$
- Constants treated same way

Fun Rule Stays the Same

- fun rule:
  - $\{x : \tau_1\} + \Gamma |- e_1 : \tau_2$
  - $\Gamma |- \text{fun } x -> e : \tau_1 \rightarrow \tau_2$
  - Types $\tau_1, \tau_2$ monomorphic
  - Function argument must always be used at same type in function body
Polymorphic Example

- Assume additional constants:
  - `hd : ∀α. α list -> α`
  - `tl : ∀α. α list -> α list`
  - `is_empty : ∀α. α list -> bool`
  - `:: : ∀α. α -> α list -> α list`
  - `[] : ∀α. α list`

Polymorphic Example

Show:

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

Polymorphic Example: Let Rec Rule

Show: (1) (2)

\[
\{\text{length:} α \text{ list -> int}\} \{\text{length:} ∀\alpha. \alpha \text{ list -> int}\}
\]

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

Polymorphic Example (1)

Show:

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

Polymorphic Example (1): Fun Rule

Show: (3)

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

Polymorphic Example (3)

Let \( \Gamma = \{\text{length:} α \text{ list -> int, l:} α \text{ list}\} \)

Show:

\[
\Gamma |- \text{if is_empty l then 0} \\
\text{else 1 + length (tl l) : int}
\]
Polymorphic Example (3): IfThenElse

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

Show

(4)  \[\Gamma |- \text{is_empty } l\]
(5)  \[\Gamma |- 0: \text{int}\]
(6)  \[\Gamma |- 1 + \text{length (tl } l) : \text{int}\]

\[\Gamma |- \text{if is_empty } l \text{ then } 0 \text{ else } 1 + \text{length (tl } l) : \text{int}\]

10/3/17  67

Polymorphic Example (4)

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

Show

\[?\]
\[\Gamma |- \text{is_empty } l : \text{bool}\]

10/3/17  68

Polymorphic Example (4): Application

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

\[\Gamma |- \text{is_empty : } \alpha \text{ list } \rightarrow \text{bool}\]
\[\Gamma |- l : \alpha \text{ list}\]
\[\Gamma |- \text{is_empty } l : \text{bool}\]

10/3/17  69

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

\[\Gamma |- \text{is_empty : } \alpha \text{ list } \rightarrow \text{bool}\]
\[\Gamma |- l : \alpha \text{ list}\]
\[\Gamma |- \text{is_empty } l : \text{bool}\]

This finishes (4)

10/3/17  71

Polymorphic Example (5): Const

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

Show

By Const Rule

\[\Gamma |- 0: \text{int}\]

10/3/17  72
Polymorphic Example (6): Arith Op

Let \( \Gamma = \{ \text{length: } \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

- By Variable
  \[ \Gamma |- \text{length} \]
- By Const
  \[ \Gamma |- (\text{tl } l) : \alpha \text{ list} \]
  \[ \Gamma |- 1: \text{int} \]
  \[ \Gamma |- \text{length} (\text{tl } l) : \text{int} \]
  \[ \Gamma |- 1 + \text{length} (\text{tl } l) : \text{int} \]

10/3/17

Polymorphic Example (7): App Rule

Let \( \Gamma = \{ \text{length: } \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

- By Const
  \[ \Gamma |- (\text{tl } l) : \alpha \text{ list } \rightarrow \alpha \text{ list} \]
  \[ \Gamma |- l : \alpha \text{ list} \]

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

10/3/17

Polymorphic Example: (2) by ArithOp

Let \( \Gamma' = \{ \text{length: } \alpha \cdot \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

\[
\begin{align*}
\text{length} ((\text{::}) 2 \, []) & : \text{int} \\
\text{length} ((\text{::}) \text{true} \, []) & : \text{int}
\end{align*}
\]

\[
\begin{align*}
\Gamma' |- \text{length} ((\text{::}) 2 \, []) + \text{length} ((\text{::}) \text{true} \, []) : \text{int}
\end{align*}
\]

10/3/17

Polymorphic Example: (8) AppRule

Let \( \Gamma' = \{ \text{length: } \forall \alpha. \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

- By Var since \( \forall \alpha. \alpha \text{ list } \rightarrow \text{int} \)
- By Const since \( \forall \alpha. \alpha \text{ list } \rightarrow \text{int} \)

10/3/17

Polymorphic Example: (10) AppRule

Let \( \Gamma' = \{ \text{length: } \forall \alpha. \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

- By Const since \( \forall \alpha. \alpha \text{ list } \rightarrow \text{int} \)

10/3/17
Polymorphic Example: (11) AppRule
- Let $\Gamma' = \{\text{length}: \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
  - By Const since $\alpha$ list is instance of $\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$

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

By Const
$$
\frac{}{\Gamma' \vdash (\cdot \cdot) 2 : \text{int list} \rightarrow \text{int list}}
$$

10/3/17

---

Polymorphic Example: (9) AppRule
- Let $\Gamma' = \{\text{length}: \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
  - By Var since $\text{bool list} \rightarrow \text{int}$ is instance of $\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$

$$
\frac{}{\Gamma' \vdash \text{length}: \text{bool list} \rightarrow \text{int}}
$$

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

10/3/17

---

Polymorphic Example: (12) AppRule
- Let $\Gamma' = \{\text{length}: \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
  - By Const since $\alpha$ list is instance of $\forall \alpha. \alpha \text{ list}$

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

By Const
$$
\frac{}{\Gamma' \vdash \text{true} : \text{bool}}
$$

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

10/3/17

---

Polymorphic Example: (13) AppRule
- Let $\Gamma' = \{\text{length}: \alpha. \alpha \text{ list} \rightarrow \text{int}\}$
- Show:
  - By Const since $\text{bool list} \rightarrow \text{int}$ is instance of $\forall \alpha. \alpha \text{ list}$

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

By Const
$$
\frac{}{\Gamma' \vdash \text{true} : \text{bool}}
$$

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

10/3/17