Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

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

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

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

Types as Specifications

- Types describe **properties**
- Different type systems describe different properties:
  - 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

- 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

- 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, JavaScript) 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
  - Haskel, OCAML, SML all use type inference
    - Records are a problem for type inference

Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \mid\!- \text{exp} : \tau$$

- 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 $\{ x : \sigma , \ldots \}$
  - 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

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

$$\frac{}{\Gamma \mid\!- \; \text{true} : \text{bool}} \qquad \frac{}{\Gamma \mid\!- \; \text{false} : \text{bool}}$$

- 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 exits, its unique

Variable axiom:

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

Simple Rules - Arithmetic

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

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

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

$$\frac{\Gamma \mid\!- e_1 : \tau \qquad \Gamma \mid\!- e_2 : \tau}{\Gamma \mid\!- e_1 \sim e_2 : \mathrm{bool}}$$

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?

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

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)

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

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;\&\&\; e_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;||\; e_2 : \text{bool}}$$

Type Variables in Rules

- If_then_else rule:

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

- 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:

$$\frac{\Gamma \;|\!\!-\; e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \;|\!\!-\; e_2 : \tau_1}{\Gamma \;|\!\!-\; (e_1\ e_2) : \tau_2}$$

- 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:

$$\frac{\{x : \tau_1\} + \Gamma \text{ |- } e : \tau_2}{\Gamma \text{ |- fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

Fun Examples

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

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

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

(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:

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

Example

- let rec rule:

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

- Which rule do we apply?

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

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**

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

Proof of 1

- Application rule:

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

- Which rule?

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

Proof of 1

- Application rule:

$$\frac{\Gamma \mid\!\!- \; e_1 : \tau_1 \to \tau_2 \quad \Gamma \mid\!\!- \; e_2 : \tau_1}{\Gamma \mid\!\!- \; (e_1\ e_2) : \tau_2}$$

- Application

$$\frac{\textcircled{3}\ \{\text{one} : \text{int list}\} \mid\!\!- \; ((::)\ 1) : \textbf{int list} \to \textbf{int list} \qquad \textcircled{4}\ \{\text{one} : \text{int list}\} \mid\!\!- \; \text{one} : \textbf{int list}}{\{\text{one} : \text{int list}\} \mid\!\!- \; (1 :: \text{one}) : \textbf{int list}}$$

Proof of 3

Constants Rule Constants Rule

$$\dfrac{\dfrac{}{\{\text{one} : \text{int list}\} \mid\!- \; (::) : \textbf{int} \to \textbf{int list} \to \textbf{int list}} \qquad \dfrac{}{\{\text{one} : \text{int list}\} \mid\!- \; 1 : \textbf{int}}}{\{\text{one} : \text{int list}\} \mid\!- \; ((::)\ 1) : \textbf{int list} \to \textbf{int list}}$$

Proof of 4

- Rule for variables

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

Proof of 2

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

- Constant

fun y ->

(x :: y :: one))

$$\frac{\{\text{one : int list}\} \text{ |- } 2 : \textbf{int} \qquad \{\text{x:int; one : int list}\} \text{ |- fun y -> (x :: y :: one))} : \textbf{int} \to \textbf{int list}}{\{\text{one : int list}\} \text{ |- (let x = 2 in fun y -> (x :: y :: one))} : \textbf{int} \to \textbf{int list}}$$

{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

?

{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 7

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

Proof of 6

Constant

Variable

$$\dfrac{\dfrac{}{\{\ldots\} \mid\!- \ (::) : \textbf{int} \rightarrow \textbf{int list} \rightarrow \textbf{int list}} \quad \dfrac{}{\{\ldots;\ x{:}int;\ldots\} \mid\!- \ \textbf{x:int}}}{\{y{:}int;\ x{:}int;\ one : int\ list\} \mid\!- \ ((::)\ x) : \textbf{int list} \rightarrow \textbf{int list}}$$

Proof of 7

Like Pf of 6 [replace x w/ y] Variable

$\vdots$

{y:int; …} |- ((::) y) :**int list→ int list**

{…; one: 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 $\mathrm{FreeVars}(\tau)$
- Free variables of polymorphic type removes variables that are universally quantified
  - $\mathrm{FreeVars}(\forall \alpha_1, \ldots, \alpha_n . \tau) = \mathrm{FreeVars}(\tau) - \{\alpha_1, \ldots, \alpha_n\}$
- $\mathrm{FreeVars}(\Gamma)$ = 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
- $Gen(\tau, \Gamma) = \forall\ \alpha_1, \ldots, \alpha_n\ .\ \tau$ where

  $\{\alpha_1, \ldots, \alpha_n\} = freeVars(\tau) - freeVars(\Gamma)$

Polymorphic Typing Rules

- A *type judgement* has the form

$$\Gamma \text{ |- exp} : \tau$$

  - **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:

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

- let rec rule:

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

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:

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

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

Polymorphic Example

- Assume additional **constants**:
- 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 (Repeat)

- Show: **(1)** **(2)**

(1)

$\{\text{length}:\alpha \text{ list} \rightarrow \text{int}\}$ |- fun lst -> … : **α list -> int**

(2)

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

{} |- let rec length =

fun lst -> if is_empty lst then 0

else 1 + length (tl lst)

in

length ((::) 2 []) + length((::) true []) **: int**

Polymorphic Example (1)

- Show:

?

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

Polymorphic Example (1): Fun Rule

- Show: (3)

{length:α list -> int, lst: α list } |-

if is_empty lst then 0

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

{length:α list -> int} |-

fun lst -> if is_empty lst then 0

else 1 + length (tl lst)

: α list -> int

Polymorphic Example (3)

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

$$\frac{?}{\Gamma \mid\!- \text{ if is\_empty l then 0 else 1 + length (tl lst)} \ : \textbf{int}}$$

Polymorphic Example (3):IfThenElse

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

(4)	(5)	(6)
Γ|- is_empty lst **: bool**	Γ|- 0:int	Γ|- 1 + length (tl lst) **: int**

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

Polymorphic Example (4)

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

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

Polymorphic Example (4):Application

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

$$\dfrac{\dfrac{?}{\Gamma \mid\!- \text{is\_empty} : \alpha \text{ list} \to \text{bool}} \qquad \dfrac{?}{\Gamma \mid\!- \text{lst} : \alpha \text{ list}}}{\Gamma \mid\!- \text{is\_empty lst} : \text{bool}}$$

Polymorphic Example (4)

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

By Const since α **list -> bool** is

instance of $\forall\alpha$. α **list -> bool** ?

$$\dfrac{\dfrac{}{\Gamma \mid\!- \text{is\_empty} : \alpha\ \textbf{list -> bool}} \qquad \dfrac{}{\Gamma \mid\!- \text{lst} : \alpha\ \textbf{list}}}{\Gamma \mid\!- \text{is\_empty lst} : \textbf{bool}}$$

Polymorphic Example (4)

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

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

By Variable Γ(lst) = α list

$$\dfrac{\dfrac{}{\Gamma \vdash \text{is\_empty} : \alpha\ \text{list} \to \text{bool}} \qquad \dfrac{}{\Gamma \vdash \text{lst} : \alpha\ \text{list}}}{\Gamma \vdash \text{is\_empty lst} : \text{bool}}$$

- This finishes (4)

Polymorphic Example (3):IfThenElse (Repeat)

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

$$\frac{\begin{array}{ccc} (4)\ \checkmark & (5) & (6) \\ \Gamma \mid\!- \text{is\_empty lst} & \Gamma \mid\!- \text{0:int} & \Gamma \mid\!- \text{1 + length (tl lst)} \\ \textbf{: bool} & & \textbf{: int} \end{array}}{\begin{array}{l} \Gamma \mid\!- \text{if is\_empty l then 0} \\ \qquad \text{else 1 + length (tl lst)}\ \textbf{: int} \end{array}}$$

Polymorphic Example (5):Const

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

By Const Rule

$$\frac{}{\Gamma \mid\!\text{-}\ 0:\textbf{int}}$$

Polymorphic Example (6):Arith Op

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

$$
\dfrac{\dfrac{}{\Gamma \text{|- 1 : } \textbf{int}}\text{By Const} \qquad \dfrac{\dfrac{\text{By Variable}}{\Gamma \text{|- length } \mathbf{: \alpha\ list \text{ -> } int}} \qquad \Gamma \text{|- (tl lst) : } \boldsymbol{\alpha}\ \textbf{list}\ (7)}{\Gamma \text{ |- length (tl lst) : } \textbf{int}}}{\Gamma \text{|- 1 + length (tl lst) : } \textbf{int}}
$$

Polymorphic Example (7):App Rule

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

$$\dfrac{\dfrac{\text{By Const}}{\Gamma \mid\!- \text{(tl lst)} : \boldsymbol{\alpha}\ \textbf{list -> }\boldsymbol{\alpha}\ \textbf{list}} \qquad \dfrac{\text{By Variable}}{\Gamma \mid\!- \text{lst} : \boldsymbol{\alpha}\ \textbf{list}}}{\Gamma \mid\!- \text{(tl lst)} : \boldsymbol{\alpha}\ \textbf{list}}$$

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

Polymorphic Example: Let Rec Rule (Repeat)

- Show: (1) ✓ (2)

$$\frac{\{\text{length}:\alpha \text{ list} \to \text{int}\} \vdash \text{fun } l \to \ldots : \alpha \text{ list} \to \text{int} \qquad \{\text{length}: \forall \alpha.\ \alpha \text{ list} \to \text{int}\} \vdash \text{length } ((::)\ 2\ [\,]) + \text{length}((::)\ \text{true}\ [\,]) : \text{int}}{\{\} \vdash \text{let rec length} = \text{fun } l \to \text{if is\_empty } l \text{ then } 0 \text{ else } 1 + \text{length } (\text{tl } l) \text{ in length } ((::)\ 2\ [\,]) + \text{length}((::)\ \text{true}\ [\,]) : \text{int}}$$

Polymorphic Example: (2) by ArithOp

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

$$\frac{\begin{array}{ll}(8) & (9) \\ \Gamma' \ \vert\!- & \Gamma' \ \vert\!- \\ \quad \text{length ((::) 2 []) :}\textbf{int} & \text{length((::) true []) : }\textbf{int}\end{array}}{\begin{array}{l}\{\text{length: } \alpha.\ \alpha \text{ list -> int}\} \\ \vert\!- \text{ length ((::) 2 []) + length((::) true []) : }\textbf{int}\end{array}}$$

Polymorphic Example: (8)AppRule

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

$$\frac{\Gamma' \;\text{|-}\; \text{length} : \textbf{int list ->int} \qquad \Gamma' \;\text{|-}\; ((::)2\;[]) : \textbf{int list}}{\Gamma' \;\text{|-}\; \text{length}\;((::)\;2\;[]) : \textbf{int}}$$

Polymorphic Example: (8)AppRule

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

By Var since int list -> int is instance of

$\forall\alpha.\ \alpha$ list -> int

$$\dfrac{\dfrac{}{\Gamma' \text{ |- length : } \textbf{int list ->int}} \qquad \overset{(10)}{\Gamma' \text{ |- ((::)2 []):} \textbf{int list}}}{\Gamma' \text{ |- length ((::) 2 []) : } \textbf{int}}$$

Polymorphic Example: (10)AppRule

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

$$\dfrac{\Gamma' \text{ |-((::) 2) : } \textbf{int list -> int list} \qquad \dfrac{(11)}{\Gamma' \text{ |- [] : } \textbf{int list}}}{\Gamma' \text{ |- ((::) 2 []) : } \textbf{int list}}$$

Polymorphic Example: (11)AppRule

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

$$\dfrac{\dfrac{\forall\alpha.\ \alpha \text{ list}}{\Gamma' \ |\text{-}\ (::) : \textbf{int -> int list -> int list}} \quad \dfrac{\text{By Const}}{\Gamma' \ |\text{-}\ 2 : \textbf{int}}}{\Gamma' \ |\text{-}\ ((::)\ 2) : \textbf{int list -> int list}}$$

Polymorphic Example: (9)AppRule

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

$$\frac{\Gamma' \text{ |- length:}\textbf{bool list ->int} \qquad \Gamma' \text{ |- ((::) true []):}\textbf{bool list}}{\Gamma' \text{ |- length ((::) true []) :}\textbf{int}}$$

Polymorphic Example: (9)AppRule

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

By Var since bool list -> int is instance of

$\forall\alpha$. α list -> int

(12)

$$\frac{\dfrac{}{\Gamma' \mid\!- \text{length} : \textbf{bool list ->int}} \qquad \dfrac{(12)}{\Gamma' \mid\!- ((::) \text{ true } []) : \textbf{bool list}}}{\Gamma' \mid\!- \text{length } ((::) \text{ true } []) : \textbf{int}}$$

Polymorphic Example: (12)AppRule

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

$$\dfrac{\Gamma' \text{ |-((::)true):}\mathbf{bool\ list\ ->bool\ list} \qquad \dfrac{(13)}{\Gamma' \text{ |- []:}\mathbf{bool\ list}}}{\Gamma' \text{ |- ((::) true []) :}\mathbf{bool\ list}}$$

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

$$\dfrac{\dfrac{\text{By Const since bool list is instance of } \forall\alpha.\ \alpha \text{ list}}{\Gamma' \mid\!- \ (::):\textbf{bool -> bool list -> bool list}} \qquad \dfrac{\text{By Const}}{\Gamma' \mid\!- \ \text{true} : \textbf{bool}}}{\Gamma' \mid\!- \ ((::)\ \text{true}) : \textbf{bool list -> bool list}}$$