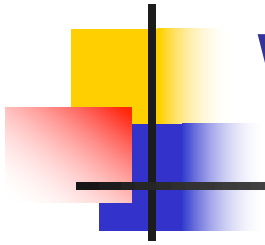
Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

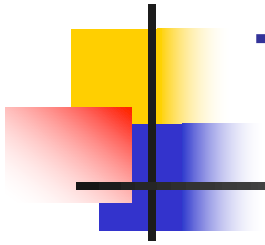
http://courses.engr.illinois.edu/cs421

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



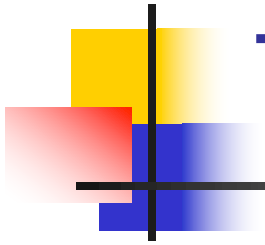
Why Data Types?

- 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



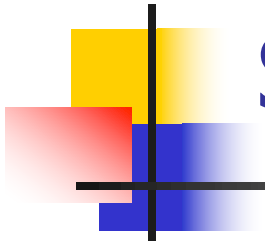
Terminology

- 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

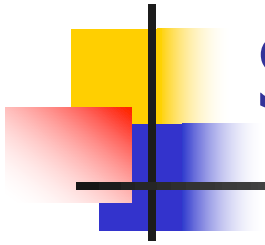
- Types describe properties
- Different type systems describe different properties, eg
  - Data is read-write versus read-only
  - Operation has authority to access data
  - Data came from “right” source
  - Operation might or could not raise an exception
- Common type systems focus on types describing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



Sound Type System

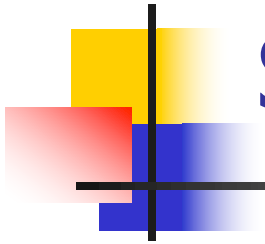
- If an expression is assigned type *t*, and it evaluates to a value *v*, then *v* is in the set of values defined by *t*

- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not



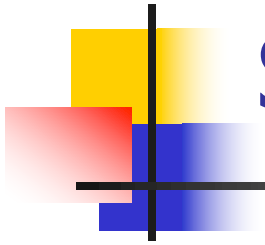
Strongly Typed Language

- 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”



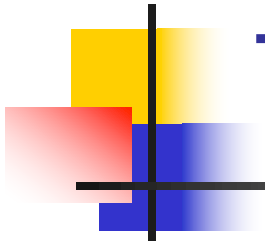
Strongly Typed Language

- 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



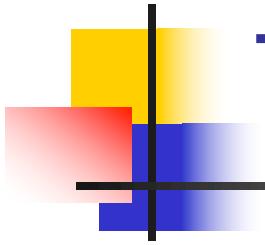
Static vs Dynamic Types

- *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



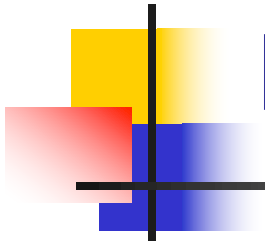
Type Checking

- 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



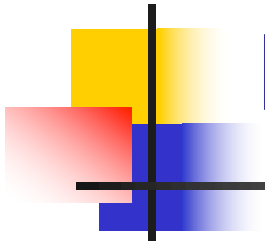
Type Checking

- 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



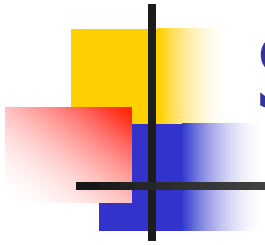
Dynamic Type Checking

- 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



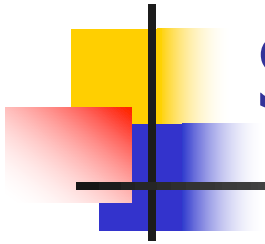
Dynamic Type Checking

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



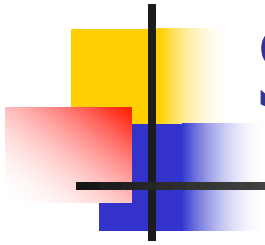
Static Type Checking

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



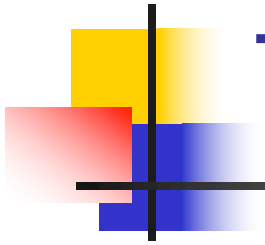
Static Type Checking

- 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



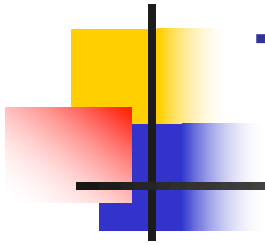
Static Type Checking

- 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

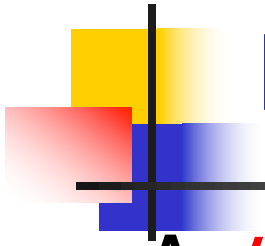
- *Type declarations*: explicit assignment of types to variables (signatures to functions) in the code of a program
  - Must be checked in a strongly typed language
  - Often not necessary for strong typing or even static typing (depends on the type system)



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

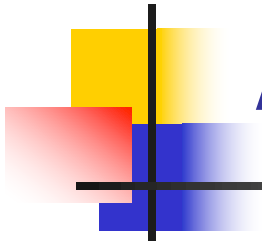
Format of Type Judgments



- A *type judgement* has the form

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

- 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”)

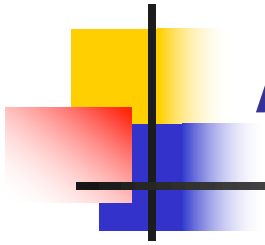


Axioms - Constants

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

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

- 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 exits, its unique

Variable axiom:

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

Simple Rules - Arithmetic

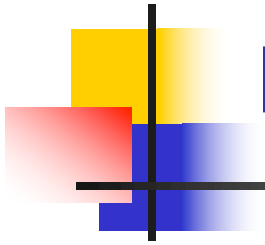
Primitive Binary operators ($\oplus \in \{ +, -, *, \ldots \}$):

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

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

$$\frac{\Gamma \mid\!- e_1 : \tau \quad \Gamma \mid\!- e_2 : \tau \quad (\sim) : \tau \rightarrow \tau \rightarrow \text{bool}}{\Gamma \mid\!- e_1 \sim e_2 : \text{bool}}$$

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?

$$\frac{\{x : int\} \vdash x + 2 : int \qquad \{x:int\} \vdash 3 : int}{\{x:int\} \vdash x + 2 = 3 : bool}\ Bin$$

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

How to finish?

$$\dfrac{\dfrac{\{x:int\} \vdash x:int \quad \{x:int\} \vdash 2:int}{\{x:int\} \vdash x + 2 : int}\,Bin \qquad \{x:int\} \vdash 3 :int}{\{x:int\} \vdash x + 2 = 3 : bool}\,Bin$$

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

Complete Proof (type derivation)

$$\dfrac{\dfrac{\dfrac{}{\{x:int\} \vdash x:int}\,Var \quad \dfrac{}{\{x:int\} \vdash 2:int}\,Const}{\{x:int\} \vdash x + 2 : int}\,Bin \quad \dfrac{}{\{x:int\} \vdash 3 :int}\,Const}{\{x:int\} \vdash x + 2 = 3 : bool}\,Bin$$

Simple Rules - Booleans

Connectives

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

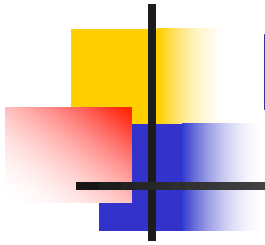
$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \text{bool}}$$

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

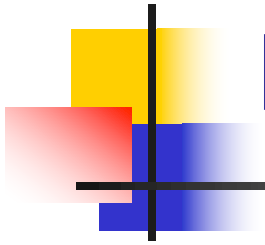


Function Application

- Application rule:

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

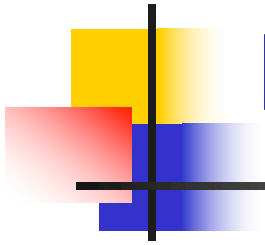
- If you have a function expression e_1 of type $\tau_1 \to \tau_2$ applied to an argument e_2 of type τ_1, the resulting expression $e_1 e_2$ has type τ_2



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

$$\frac{\{y : int\} + \Gamma \vdash y + 3 : int}{\Gamma \vdash fun\ y \rightarrow y + 3 : int \rightarrow int}$$

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

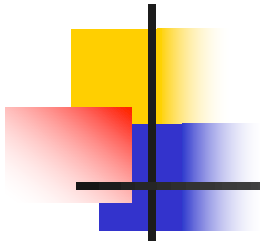
(Monomorphic) Let and Let Rec

- let rule:

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

- 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

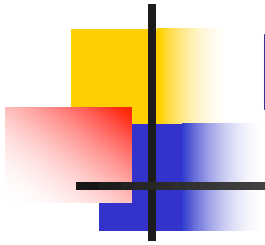
- Which rule do we apply?

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

Example

- Let rec rule:

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



Proof of 1

- Which rule?

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

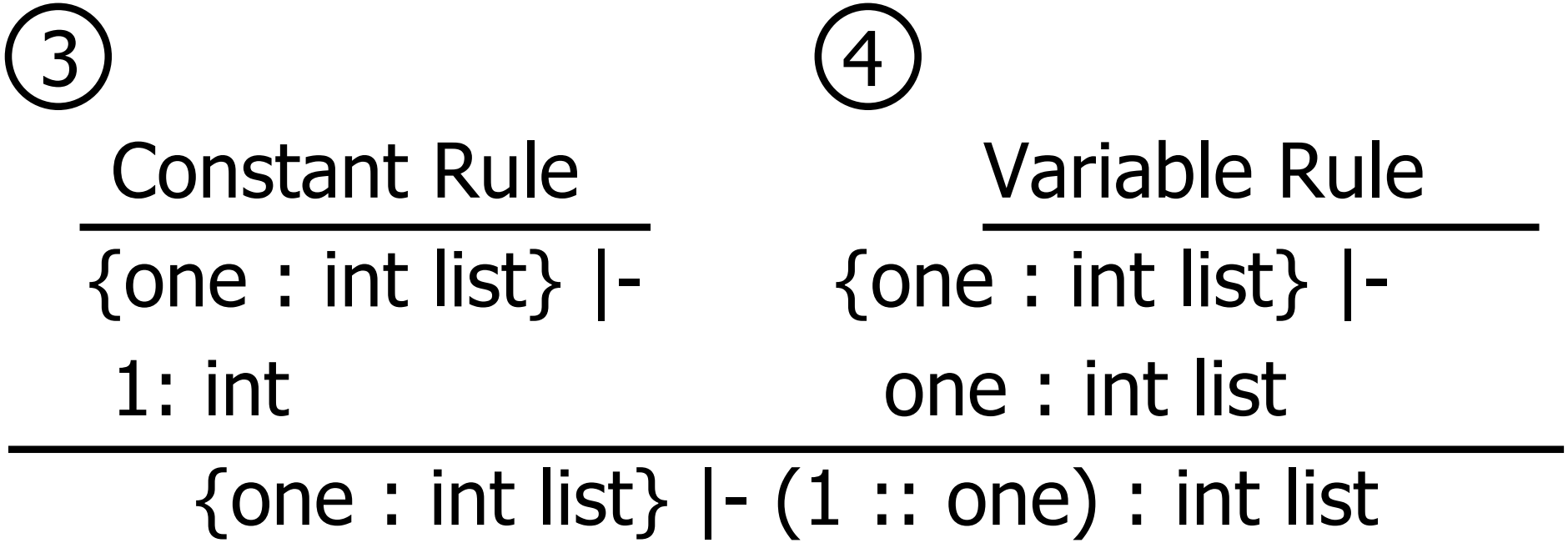
Proof of 1

- Binary Operator

$$\frac{\begin{array}{cc} \textcircled{3} & \textcircled{4} \\ \{one : int\ list\} \vdash & \{one : int\ list\} \vdash \\ 1 : int & one : int\ list \end{array}}{\{one : int\ list\} \vdash (1 :: one) : int\ list}$$

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

Proof of 1


③
④
Constant Rule
Variable Rule
{one : int list} |-
1: int
{one : int list} |-
one : int list
{one : int list} |- (1 :: one) : int list

Proof of 2

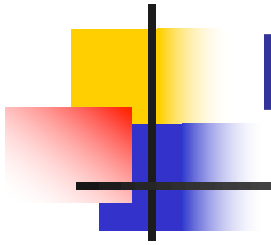
- Let Rule

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

Proof of 2

- Constant

$$\frac{\dfrac{}{\{one : int\ list\} \vdash 2 : int} \qquad \textcircled{5}\ \dfrac{}{\{x:int;\ one : int\ list\} \vdash \begin{array}{l}fun\ y \rightarrow \\ \quad (x :: y :: one)) \\ \quad : int \rightarrow int\ list\end{array}}}{\{one : int\ list\} \vdash \begin{array}{l}(let\ x = 2\ in \\ \quad fun\ y \rightarrow (x :: y :: one)) : int \rightarrow int\ list\end{array}}$$



Proof of 5

?

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

: int $\rightarrow$ int list

Proof of 5

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

By the Fun Rule

Proof of 5

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

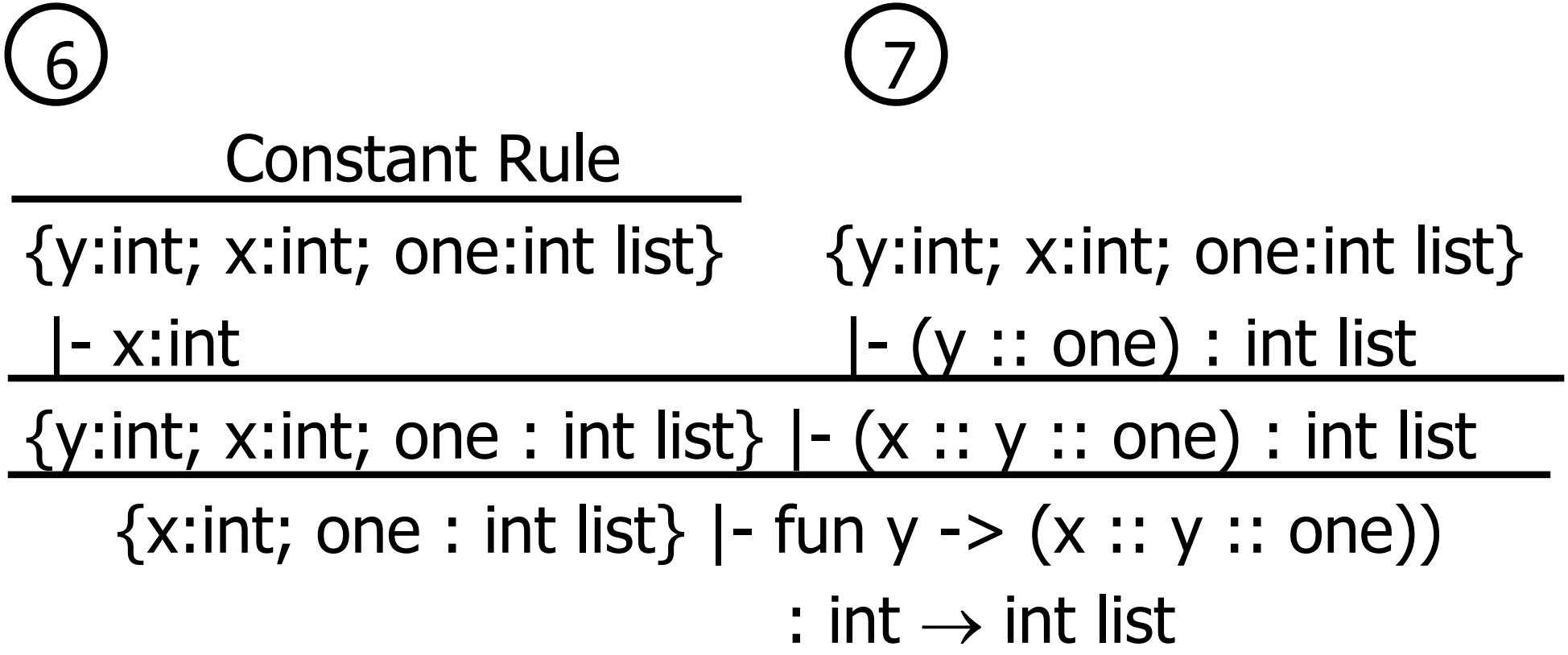
(7) {y:int; x:int; one:int list} |- (y :: one) : int list

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

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

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

Proof of 6


6
7
Constant Rule
{y:int; x:int; one:int list}
|- x:int
{y:int; x:int; one:int list}
|- (y :: one) : int list
{y:int; x:int; one : int list} |- (x :: y :: one) : int list
{x:int; one : int list} |- fun y -> (x :: y :: one))
: int → int list

Proof of 7

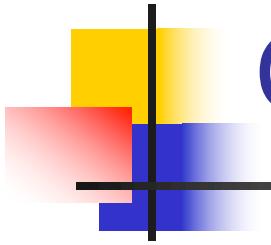
- Binary Operation Rule

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

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

Proof of 7

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



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

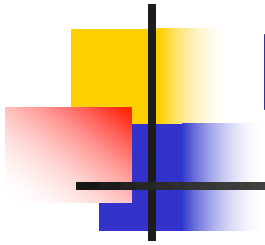
Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

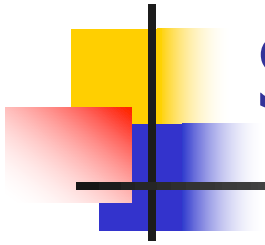
- Application

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



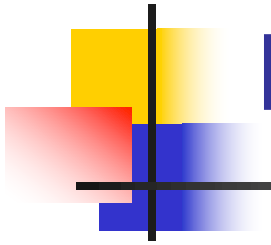
Mea Culpa

- 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



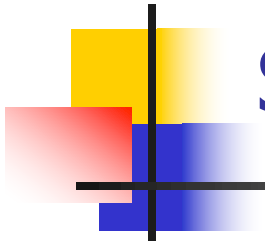
Support for Polymorphic Types

- 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 $\forall \,.\, \tau$



Example FreeVars Calculations

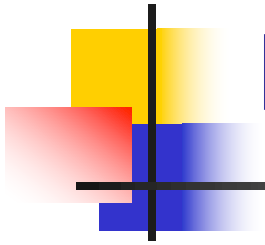
- Vars(‘a -> (int -> ‘b) -> ‘a) ={‘a , ‘b}
- FreeVars (All ‘b. ‘a -> (int -> ‘b) -> ‘a) =
- {‘a , ‘b} – {‘b}= {‘a}
- FreeVars {x : All ‘b. ‘a -> (int -> ‘b) -> ‘a,
- id: All ‘c. ‘c -> ‘c,
- y: All ‘c. ‘a -> ‘b -> ‘c} =
- {‘a} U {} U {‘a, ‘b} = {‘a, ‘b}



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

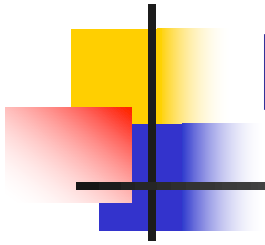
Monomorphic to Polymorphic



- 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
- $Gen(\tau, \Gamma) = \forall \alpha_1, \ldots, \alpha_n . \tau$ where

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

Polymorphic Typing Rules



- A *type judgement* has the form

  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

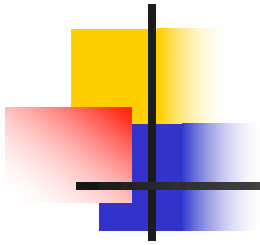
Polymorphic Let and Let Rec

- let rule:

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

- let rec rule:

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



Polymorphic Variables (Identifiers)

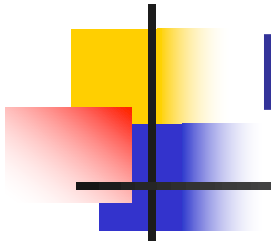
Variable axiom:

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

- 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 and primitive operators:
- 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:

$$\frac{?}{\{\} \vdash \texttt{let rec length = fun l -> if is\_empty l then 0 else 1 + length (tl l) in length (2 :: []) + length(true :: []) : int}}$$

Polymorphic Example: Let Rec Rule

- Show: (1) (2)

{length:α list -> int} {length:$\forall\alpha$. α list -> int}

|- fun l -> … |- length (2 :: []) +

: α list -> int 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

$$\frac{?}{\Gamma \vdash \text{if is\_empty l then 0 else 1 + length (tl l)} : \text{int}}$$

Polymorphic Example (3):IfThenElse

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

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

Polymorphic Example (4)

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

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

Polymorphic Example (4):Application

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

$$\frac{\dfrac{?}{\Gamma \vdash \text{is\_empty} : \alpha \text{ list} \rightarrow \text{bool}} \qquad \dfrac{?}{\Gamma \vdash \text{l} : \alpha \text{ list}}}{\Gamma \vdash \text{is\_empty l} : \text{bool}}$$

Polymorphic Example (4)

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

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

$$\frac{\Gamma \vdash \text{is\_empty} : \alpha \text{ list} \to \text{bool} \qquad \Gamma \vdash l : \alpha \text{ list}}{\Gamma \vdash \text{is\_empty } l : \text{bool}}$$

Polymorphic Example (4)

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

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

By Variable $\Gamma(l) = \alpha$ list

$$\frac{\Gamma \vdash \text{is\_empty} : \alpha \text{ list} \to \text{bool} \qquad \Gamma \vdash l : \alpha \text{ list}}{\Gamma \vdash \text{is\_empty } l : \text{bool}}$$

- This finishes (4)

Polymorphic Example (5):Const

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

By Const Rule

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

Polymorphic Example (6):Arith Op

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

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

Polymorphic Example (7):App Rule

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

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

By Const since α list -> α list is instance of

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

Polymorphic Example: (2) by ArithOp

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

(8) (9)

Γ' |- Γ' |-

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

{length:$\forall\alpha.\ \alpha$ list -> int}

|- length (2 :: []) + length(true :: []) : int

Polymorphic Example: (8)AppRule

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

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