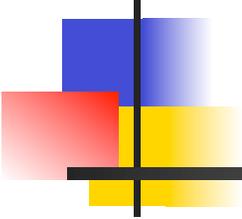


# Programming Languages and Compilers (CS 421)



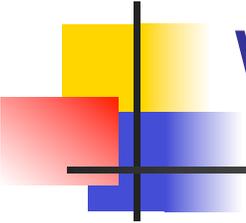
---

Elsa L Gunter

2112 SC, UIUC

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

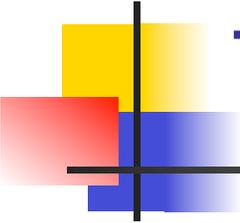
Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



# Why Data Types?

---

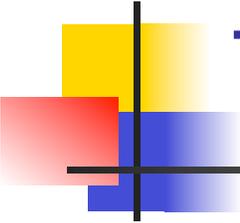
- Data types play a key role in:
  - *Data abstraction* in the design of programs
  - *Type checking* in the analysis of programs
  - *Compile-time code generation* in the translation and execution of programs
    - Data layout (how many words; which are data and which are pointers) dictated by type



# Terminology

---

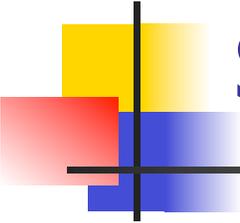
- Type: A **type**  $t$  defines a set of possible data values
  - E.g. **short** in C is  $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
  - A value in this set is said to have type  $t$
- Type system: rules of a language assigning types to expressions



# Types as Specifications

---

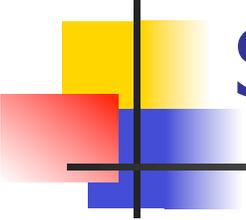
- Types describe properties
- Different type systems describe different properties, eg
  - Data is read-write versus read-only
  - Operation has authority to access data
  - Data came from “right” source
  - Operation might or could not raise an exception
- Common type systems focus on types describing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



# Sound Type System

---

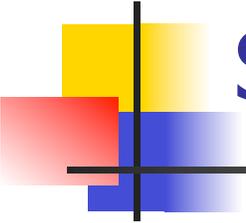
- If an expression is assigned type  $t$ , and it evaluates to a value  $v$ , then  $v$  is in the set of values defined by  $t$
- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not



# Strongly Typed Language

---

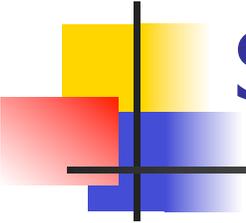
- When no application of an operator to arguments can lead to a run-time type error, language is *strongly typed*
  - Eg: `1 + 2.3;;`
- Depends on definition of “type error”



# Strongly Typed Language

---

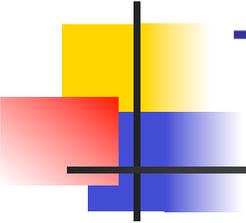
- C++ claimed to be “strongly typed”, but
  - Union types allow creating a value at one type and using it at another
  - Type coercions may cause unexpected (undesirable) effects
  - No array bounds check (in fact, no runtime checks at all)
- SML, OCAML “strongly typed” but still must do dynamic array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



# Static vs Dynamic Types

---

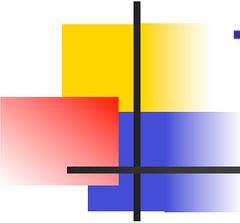
- *Static type*: type assigned to an expression at compile time
- *Dynamic type*: type assigned to a storage location at run time
- *Statically typed language*: static type assigned to every expression at compile time
- *Dynamically typed language*: type of an expression determined at run time



# Type Checking

---

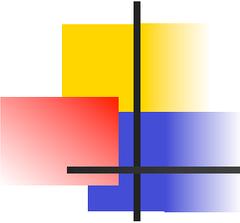
- When is  $op(arg1, \dots, argn)$  allowed?
- *Type checking* assures that operations are applied to the right number of arguments of the right types
  - Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- Used to resolve overloaded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



# Type Checking

---

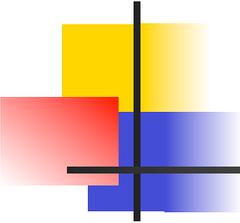
- Type checking may be done *statically* at compile time or *dynamically* at run time
- Dynamically typed (aka untyped) languages (eg LISP, Prolog) do only dynamic type checking
- Statically typed languages can do most type checking statically



# Dynamic Type Checking

---

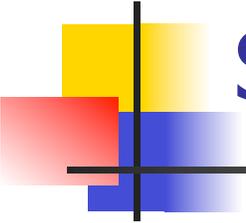
- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different types



# Dynamic Type Checking

---

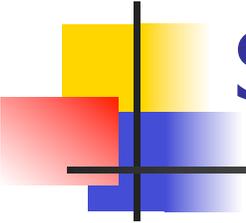
- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe years after the code was written)



# Static Type Checking

---

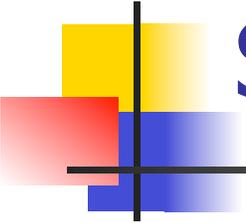
- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time



# Static Type Checking

---

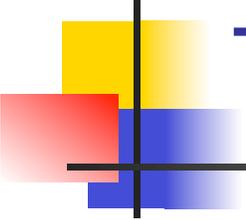
- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can't check types that depend on dynamically computed values
  - Eg: array bounds



# Static Type Checking

---

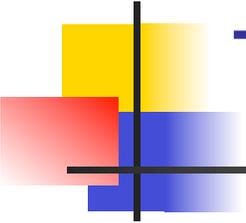
- Typically places restrictions on languages
  - Garbage collection
  - References instead of pointers
  - All variables initialized when created
  - Variable only used at one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks



# Type Declarations

---

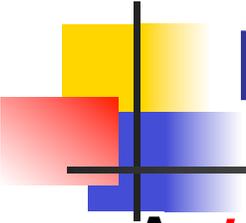
- *Type declarations*: explicit assignment of types to variables (signatures to functions) in the code of a program
  - Must be checked in a strongly typed language
  - Often not necessary for strong typing or even static typing (depends on the type system)



# Type Inference

---

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



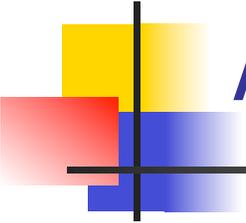
# Format of Type Judgments

---

- A *type judgement* has the form

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

- $\Gamma$  is a typing environment
  - Supplies the types of variables and functions
  - $\Gamma$  is a set of the form  $\{ x : \sigma , \dots \}$
  - 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  $\text{exp}$
- $\vdash$  pronounced “turnstile”, or “entails” (or “satisfies” or, informally, “shows”)



# Axioms - Constants

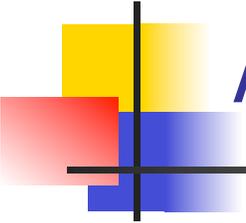
---

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

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

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

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



## Axioms – Variables (Monomorphic Rule)

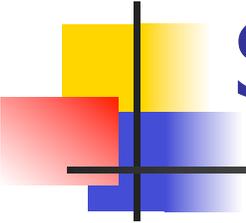
---

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

**Note:** if such  $\sigma$  exists, its unique

Variable axiom:

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



# Simple Rules - Arithmetic

---

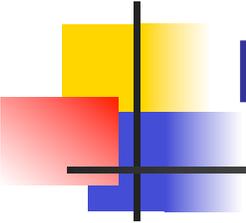
Primitive operators ( $\oplus \in \{+, -, *, \dots\}$ ):

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

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

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

For the moment, think  $\tau$  is **int**

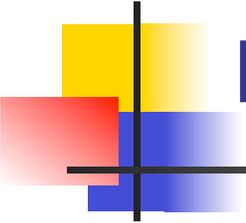


Example:  $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

---

What do we need to show first?

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

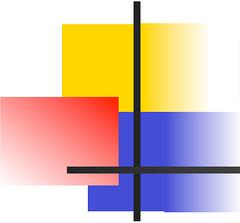


Example:  $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

---

What do we need for the left side?

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

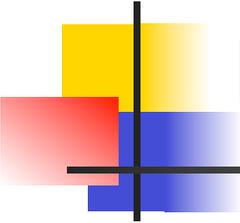


Example:  $\{x:\text{int}\} \Vdash x + 2 = 3 : \text{bool}$

---

How to finish?

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

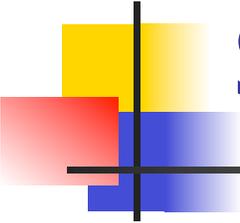


Example:  $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

---

Complete Proof (type derivation)

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



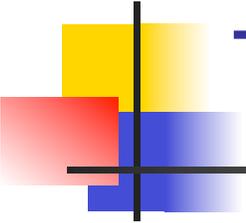
# Simple Rules - Booleans

---

## Connectives

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

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



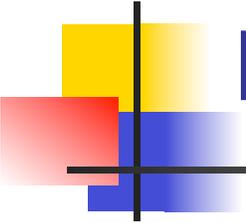
# Type Variables in Rules

---

- If\_then\_else rule:

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

- $\tau$  is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if\_then\_else must all have same type



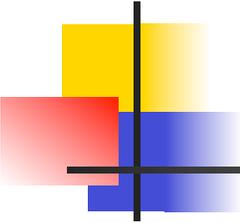
# Function Application

---

- Application rule:

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

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

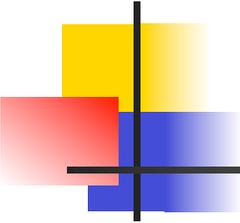


## Fun Rule

---

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

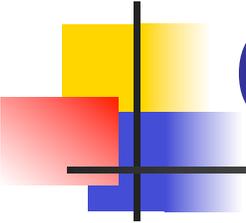
$$\frac{\{x : \tau_1\} + \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$



# Fun Examples

---

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



## (Monomorphic) Let and Let Rec

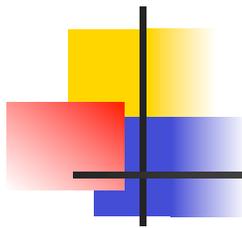
---

- let rule:

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

- let rec rule:

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



# Example

---

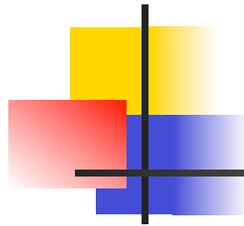
- Which rule do we apply?

?

---

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



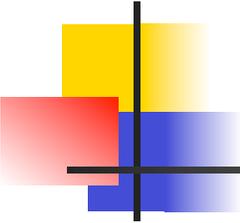


# Proof of 1

---

- Which rule?

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



# Proof of 1

---

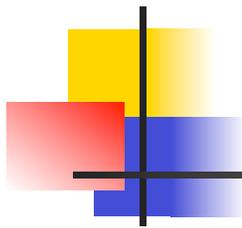
- Application

③

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

④

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



# Proof of 3

---

Constants Rule

Constants Rule

---

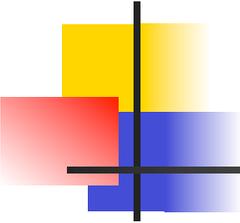
$$\{one : int\ list\} \vdash$$
$$(::) : int \rightarrow int\ list \rightarrow int\ list$$

---

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

---

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

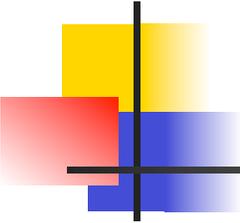


## Proof of 4

---

- Rule for variables

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



## Proof of 2

---

- Constant

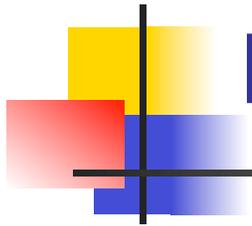
⑤  $\{x:\text{int}; \text{one} : \text{int list}\} \vdash$   
 $\text{fun } y \rightarrow$   
 $(x :: y :: \text{one}))$

---

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

---

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



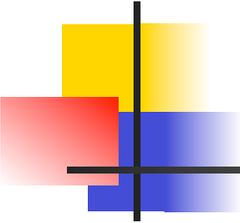
# Proof of 5

---

?

---

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



# Proof of 5

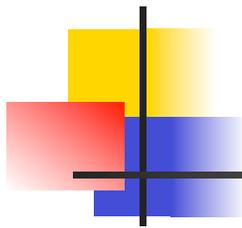
---

?

---

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





# Proof of 6

---

Constant

Variable

---


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

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

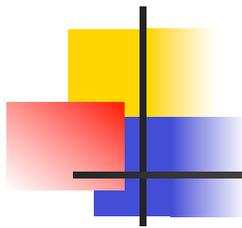

---


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


---


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

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



# Proof of 7

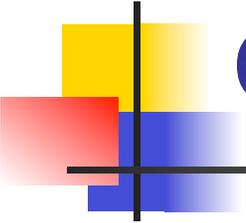
---

Pf of 6 [y/x]

Variable

•  
•  
•

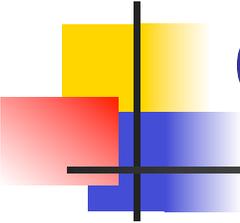
$$\frac{\frac{\{y:\text{int}; \dots\} \vdash ((::) y) : \text{int list} \rightarrow \text{int list}}{\{y:\text{int}; x:\text{int}; \text{one} : \text{int list}\} \vdash (y :: \text{one}) : \text{int list}}}{\{y:\text{int}; \dots\} \vdash ((::) y) : \text{int list} \rightarrow \text{int list}} \quad \frac{\{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



# Curry - Howard Isomorphism

---

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms
  
- Function space arrow corresponds to implication; application corresponds to modus ponens



# Curry - Howard Isomorphism

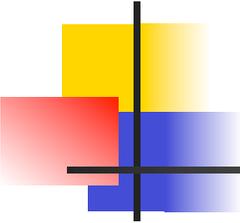
---

- Modus Ponens

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

- Application

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



# Mia Copa

---

- 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