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

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*
  - E.g: $1 + 2.3;$;
  - 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 \(\text{op(arg1,...,argn)}\) 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

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
- 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

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

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 derivation: A formal proof that a term has a type,
- assuming types for variables
- using the rules of a type system

Type checking: A program to analyze code
- Confirms terms in the code have needed types according to the type system
- Assures type derivations exist

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: 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 \( \Gamma |- \exp : \tau \)
- \( \Gamma \) 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 \( (x : \sigma \in \Gamma) \)
- \( \exp \) is a program expression
- \( \tau \) is a type to be assigned to \( \exp \)
- \( |- \) pronounced “turnstyle”, or “entails” (or “satisfies” or, informally, “shows”)

Axioms – Constants (Monomorphic)

- \( \Gamma |- n : \text{int} \) (assuming \( n \) is an integer constant)
- \( \Gamma |- \text{true} : \text{bool} \)
- \( \Gamma |- \text{false} : \text{bool} \)

- 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 (Mono)

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

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

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

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

For the moment, think \( \tau \) is \text{int}

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

What do we need to show first?

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

What do we need for the left side?

\[
\begin{align*}
\{ x : \text{int} \} |- x + 2 : \text{int} & \quad \{ x : \text{int} \} |- 3 : \text{int} \\
\{ x : \text{int} \} |- x + 2 = 3 : \text{bool} & \quad \text{Bin}
\end{align*}
\]
Example: \( \{x:\text{int}\} \vdash x + 2 = 3 : \text{bool} \)

How to finish?

\[
\bin{x} \vdash \text{int} \quad \bin{x} \vdash \text{int} \quad \bin{2} \vdash \text{int} \\
\{x : \text{int}\} \vdash x + 2 : \text{int} \\
\{x : \text{int}\} \vdash x + 2 = 3 : \text{bool}
\]

Complete Proof (type derivation)

\[
\begin{align*}
\text{Var} & \quad \text{Const} \\
\bin{x} \vdash \text{int} & \quad \bin{x} \vdash \text{int} \\
\{x : \text{int}\} \vdash x + 2 : \text{int} & \quad \{x : \text{int}\} \vdash 3 : \text{int} \\
\{x : \text{int}\} \vdash x + 2 = 3 : \text{bool} & \quad \text{Bin}
\end{align*}
\]

How to finish?

\[
\bin{x} \vdash \text{int} \quad \bin{x} \vdash \text{int} \quad \bin{2} \vdash \text{int} \\
\{x : \text{int}\} \vdash x + 2 : \text{int} \\
\{x : \text{int}\} \vdash x + 2 = 3 : \text{bool}
\]

Type Variables in Rules

- If_then_else rule:
  \[
  \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \\
  \Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau
  \]
  - \( \tau \) 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

Example derivation: if-then-else-

\[
\begin{align*}
\Gamma & = \{x:\text{int}, \text{int}_\text{of}_\text{float} : \text{float} \rightarrow \text{int}, y : \text{float}\} \\
\Gamma \vdash (\text{fun } y \rightarrow \\
y > 3) x & \quad \Gamma \vdash x + 2 \quad \Gamma \vdash \text{int}_\text{of}_\text{float} y : \text{bool} \\
\end{align*}
\]

\[
\Gamma \vdash \text{if } (\text{fun } y \rightarrow y > 3) x \\
\text{then } x + 2 \\
\text{else } \text{int}_\text{of}_\text{float} y : \text{int}
\]

Function Application

- Application rule:
  \[
  \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \\
  \Gamma \vdash (e_1 \ e_2) : \tau_2
  \]
  - 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 \( \tau_2 \)
Example: Application

\[ \Gamma = \{ x : \text{int}, \text{int}_\text{of}_\text{float} : \text{float} \to \text{int}, y : \text{float} \} \]

\[ \Gamma \vdash (\text{fun } y \to y > 3) : \text{int} \to \text{bool} \]

\[ \Gamma \vdash x : \text{int} \]

\[ \Gamma \vdash (\text{fun } y \to y > 3) x : \text{bool} \]

Fun Rule

- Rules describe types, but also how the environment \( \Gamma \) may change
- Can only do what rule allows!
- fun rule:
  \[ \frac{\{ x : \tau_1 \} + \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \to e : \tau_1 \to \tau_2} \]

Fun Examples

\[ \{ y : \text{int} \} + \Gamma \vdash y + 3 : \text{int} \]

\[ \Gamma \vdash \text{fun } y \to y + 3 : \text{int} \to \text{int} \]

\[ \{ f : \text{int} \to \text{bool} \} + \Gamma \vdash f \ 2 : [\text{true}] : \text{bool list} \]

\[ \Gamma \vdash (\text{fun } f \to (f \ 2) : [\text{true}]) : (\text{int} \to \text{bool}) \to \text{bool list} \]

(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?

\[ \{ \} \vdash (\text{let rec } \text{one} = 1 :: \text{one} \text{ in} \]

\[ \quad \text{let } x = 2 \text{ in} \]

\[ \quad \text{fun } y \to (x :: y :: \text{one}) : \text{int} \to \text{int list} \]
Proof of 1

Which rule?

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

Proof of 1

Binary Operator

\[
\begin{array}{c}
\{\text{one} : \text{int list}\} \vdash 1 : \text{int} \\
\{\text{one} : \text{int list}\} \vdash \text{one} : \text{int list}
\end{array}
\]

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

where \((::) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list}\)

Proof of 1

Constant Rule          Variable Rule

\[
\begin{array}{c}
\{\text{one} : \text{int list}\} \vdash 1 : \text{int} \\
\{\text{one} : \text{int list}\} \vdash \text{one} : \text{int list}
\end{array}
\]

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

Proof of 2

Let Rule

\[
\{x : \text{int}; \text{one} : \text{int list}\} \vdash \text{fun y -> (x :: y :: one)}
\]

\[
\{\text{one} : \text{int list}\} \vdash 2 : \text{int} : \text{int} \rightarrow \text{int list}
\]

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

Proof of 2

Constant

\[
\{x : \text{int}; \text{one} : \text{int list}\} \vdash \text{fun y -> (x :: y :: one)}
\]

\[
\{\text{one} : \text{int list}\} \vdash 2 : \text{int} : \text{int} \rightarrow \text{int list}
\]

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

Proof of 5

\[
\{x : \text{int}; \text{one} : \text{int list}\} \vdash \text{fun y -> (x :: y :: one)}
\]

\[
\vdash \text{int} \rightarrow \text{int list}
\]
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 

By the Fun Rule

Proof of 5

6

? 
{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 

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

Proof of 6

Variable Rule

6

{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 6

7

Variable Rule

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

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

Proof of 7

Binary Operation Rule

7

Variable Rule

{y:int; x:int; one:int list} |- y:int 
{y:int; x:int; 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
  \[ A \implies B \quad A \quad \rightarrow \quad B \]

- Application
  \[ \Gamma \vdash e_1 : \alpha \implies \beta \quad \Gamma \vdash e_2 : \alpha \]
  \[ \Gamma \vdash (e_1 \ e_2) : \beta \]