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

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

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

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

### 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

**Type Declarations**
- 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 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
- \( |- \) 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 \( \sigma \) exits, its unique

Variable axiom:

\[ \Gamma \vdash x : \sigma \]  if \( \Gamma(x) = \sigma \)

Simple Rules - Arithmetic

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

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad (\oplus) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \]
\[ \Gamma \vdash e_1 \oplus e_2 : \tau_3 \]

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

\[ \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad (\sim) : \tau \rightarrow \tau \rightarrow \text{bool} \]
\[ \Gamma \vdash e_1 \sim e_2 : \text{bool} \]

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

How to finish?

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

Complete Proof (type derivation)

\[
\begin{array}{c}
\text{Var} & \text{Const} \\
\{x : \text{int}\} |- x : \text{int} & \{x : \text{int}\} |- 2 : \text{int} & \text{Bin} \\
\{x : \text{int}\} |- x + 2 : \text{int} & \{x : \text{int}\} |- 3 : \text{int} & \text{Bin} \\
\{x : \text{int}\} |- x + 2 = 3 : \text{bool} \\
\end{array}
\]

10/5/21

Simple Rules - Booleans

Connectives

\[
\begin{array}{c}
\Gamma |- e_1 : \text{bool} & \Gamma |- e_2 : \text{bool} \\
\Gamma |- e_1 \&\& e_2 : \text{bool} \\
\Gamma |- e_1 || e_2 : \text{bool} \\
\end{array}
\]

10/5/21

Type Variables in Rules

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

- \(\tau\) 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

10/5/21

Function Application

- Application rule:
  \[
  \Gamma |- e_1 : \tau_1 \rightarrow \tau_2 & \Gamma |- e_2 : \tau_1 \\
  \Gamma |- (e_1 e_2) : \tau_2 \\
  \]

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

10/5/21

Fun Rule

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

10/5/21

Fun Examples

- \(\{y : \text{int}\} + \Gamma |- y + 3 : \text{int}\)
  \[
  \Gamma |- \text{fun } y -> y + 3 : \text{int} \rightarrow \text{int} \\
  \]

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

10/5/21
(Monomorphic) Let and Let Rec

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

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

Example

- Which rule do we apply?

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

Proof of 1

- Binary Operator

\[ 3 \quad \{ \text{one : int list} \} |- 4 \quad \{ \text{one : int list} \} |- 1: \text{int} \quad \text{one : int list} \quad \text{one : int list} \quad \{ \text{one : int list} \} |- (1 :: \text{one}) : \text{int list} \]

where \( (::) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list} \)
Proof of 2

Let Rule

\{x : \text{int}; \text{one} : \text{int list}\} |- \text{fun } y \rightarrow (x :: y :: \text{one})

\{\text{one} : \text{int list}\} |- 2 : \text{int} \quad : \text{int} \rightarrow \text{int list}

\{\text{one} : \text{int list}\} |- \text{(let } x = 2 \text{ in}

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

Proof of 2

Constant Rule

\{x : \text{int}; \text{one} : \text{int list}\} |- \text{fun } y \rightarrow (x :: y :: \text{one})

\{\text{one} : \text{int list}\} |- 2 : \text{int} \quad : \text{int} \rightarrow \text{int list}

\{\text{one} : \text{int list}\} |- \text{(let } x = 2 \text{ in}

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

Proof of 5

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

\{\text{one} : \text{int list}\} |- \text{fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}

By the Fun Rule

Proof of 5

\{x : \text{int}; y : \text{int}; \text{one} : \text{int list}\} |- \text{fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}

By BinOp where ( :: ) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list}
Proof of 7

- **Binary Operation Rule**

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

By BinOp where ( :: ) : int \rightarrow \text{int list} → \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

---

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

- **Monomorphic Types (τ):**
  - Basic Types: int, bool, float, string, unit, ...
  - Type Variables: α, β, γ, δ, ε
  - Compound Types: α → β, int * string, bool list, ...

- **Polymorphic Types:**
  - Monomorphic types τ
  - Universally quantified monomorphic types
  - ∀α₁, ..., αₙ ∃τ
  - Can think of τ as same as ∀τ
Example FreeVars Calculations

- \text{Vars}(\text{a} \to (\text{int} \to \text{b}) \to \text{a}) = \{\text{a}, \text{b}\}
- \text{FreeVars}(\text{All } \text{b}. \text{a} \to (\text{int} \to \text{b}) \to \text{a}) = \{\text{a}, \text{b}\}
- \text{FreeVars}(\{x : \text{All } \text{b}. \text{a} \to (\text{int} \to \text{b}) \to \text{a}\}) = \{\text{a}, \text{b}\}

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

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 \( \Gamma \)

Polymorphic Typing Rules

- A \textit{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:

\[ \Gamma |- x : \varphi(\tau) \quad \text{if } \Gamma(x) = \forall a_1, \ldots, a_n . \tau \]

- Where \( \varphi \) replaces all occurrences of \( a_1, \ldots, a_n \) by monotypes \( \tau_1, \ldots, \tau_n \)
- Note: Monomorphic rule special case:

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

- Constants treated same way
Fun Rule Stays the Same

- Fun rule:
  \[
  \{x: \tau_1\} + \Gamma |- e: \tau_2 \\
  \Gamma |- \text{fun } x -> e: \tau_1 \rightarrow \tau_2
  \]

- Types $\tau_1, \tau_2$ monomorphic
- Function argument must always be used at same type in function body

Polymorphic Example

- Assume additional constants and primitive operators:
  - $\text{hd} : \forall \alpha. \alpha \text{ list } \rightarrow \alpha$
  - $\text{tl} : \forall \alpha. \alpha \text{ list } \rightarrow \alpha \text{ list}$
  - $\text{is}_\text{empty} : \forall \alpha. \alpha \text{ list } \rightarrow \text{bool}$
  - $(::) : \forall \alpha. \alpha \rightarrow \alpha \text{ list } \rightarrow \alpha \text{ list}$
  - $[] : \forall \alpha. \alpha \text{ list}$

Polymorphic Example: Let Rec Rule

- Show: (1) (2)
  \[
  \{\text{length}: \alpha \text{ list } \rightarrow \text{int}\} \quad \{\text{length}: \forall \alpha. \alpha \text{ list } \rightarrow \text{int}\} \\
  |- \text{fun } l -> \ldots \\
  |- \text{length} (2 :: []) + \\
  : \alpha \text{ list } \rightarrow \text{int} \\
  \]

  \[
  \text{length(true :: []) : int} \\
  \]

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

Polymorphic Example (1)

- Show:

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

Polymorphic Example (1): Fun Rule

- Show: (3)
  \[
  \{\text{length}: \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list } \} |- \\
  \text{if is_empty } l \text{ then 0} \\
  \quad \text{else } \text{length} (\text{hd } l) + \text{length} (\text{tl } l) : \text{int}
  \]

  \[
  \{\text{length}: \alpha \text{ list } \rightarrow \text{int}\} |- \\
  \text{fun } l -> \text{if is_empty } l \text{ then 0} \\
  \quad \text{else } 1 + \text{length} (\text{tl } l) \\
  : \alpha \text{ list } \rightarrow \text{int}
  \]
Polymorphic Example (3)

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

Show

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

10/5/21

Polymorphic Example (3): IfThenElse

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

Show

<table>
<thead>
<tr>
<th></th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \text{is_empty } l$</td>
<td>$\Gamma \vdash 0 : \text{int}$</td>
<td>$\Gamma \vdash 1 + \text{length } (\text{tl } l) : \text{int}$</td>
<td>$\Gamma \vdash \text{if is_empty } l \ 	ext{then } 0 \ 	ext{else } 1 + \text{length } (\text{tl } l) : \text{int}$</td>
</tr>
</tbody>
</table>

10/7/21

Polymorphic Example (4)

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

Show

? $\Gamma \vdash \text{is_empty } : \alpha \text{-list } \to \text{bool}$

10/5/21

Polymorphic Example (4): Application

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

Show

? $\Gamma \vdash \text{is_empty } : \alpha \text{-list } \to \text{bool}$

$\Gamma \vdash l : \alpha \text{-list}$

$\Gamma \vdash \text{is_empty } l : \text{bool}$

10/5/21

Polymorphic Example (4)

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

Show

By Const since $\alpha \text{-list } \to \text{bool}$ is instance of $\forall \alpha. \alpha \text{-list } \to \text{bool}$

$\Gamma \vdash \text{is_empty } : \alpha \text{-list } \to \text{bool}$

$\Gamma \vdash l : \alpha \text{-list}$

$\Gamma \vdash \text{is_empty } l : \text{bool}$

10/5/21

This finishes (4)
Polymorphic Example (5): Const

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

Show

By Const Rule

$$\Gamma |- 0: \text{int}$$

Polymorphic Example (6): Arith Op

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

Show

By Variable

$$\Gamma |- \text{length}$$  

By Const

$$\Gamma |- l: \alpha \text{ list}$$

$$\Gamma |- (\text{tl} \ l): \alpha \text{ list}$$

$$\Gamma |- 1: \text{int}$$

$$\Gamma |- \text{length} (\text{tl} \ l): \text{int}$$

$$\Gamma |- 1 + \text{length} (\text{tl} \ l): \text{int}$$

Polymorphic Example (7): App Rule

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

Show

By Const

$$\Gamma |- \text{tl : } \alpha \text{ list -> } \alpha \text{ list}$$

$$\Gamma |- l: \alpha \text{ list}$$

$$\Gamma |- (\text{tl} \ l): \alpha \text{ list}$$

By Const since $\alpha \text{ list -> } \alpha \text{ list}$ is instance of $\forall \alpha. \alpha \text{ list -> } \alpha \text{ list}$

Polymorphic Example: (2) by ArithOp

Let $\Gamma’ = \{\text{length:} \forall \alpha. \alpha \text{ list -> int}\}$

Show:

$$\Gamma’ |- \text{length} : \text{int list -> int}$$

$$\Gamma’ |- (2 :: []) : \text{int list}$$

$$\Gamma’ |- \text{length} (2 :: []) : \text{int}$$

$$\Gamma’ |- \text{length} (\text{true} :: []) : \text{int}$$

$$\Gamma’ |- \text{length} (\forall \alpha. \alpha \text{ list -> int})$$

$$\Gamma’ |- \text{length} (\text{true} :: []) : \text{int}$$

Polymorphic Example: (8) App Rule

Let $\Gamma’ = \{\text{length:} \forall \alpha. \alpha \text{ list -> int}\}$

Show:

$$\Gamma’ |- \text{length : int list -> int}$$

$$\Gamma’ |- (2 :: []) : \text{int list}$$

$$\Gamma’ |- \text{length} (2 :: []) : \text{int}$$