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 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

**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 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
  \[ \Gamma \vdash \text{exp} : \tau \]
  - \( \Gamma \) 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 \( \{ x : \alpha, \ldots \} \)
  - For any \( x \) at most one \( \alpha \) such that \( (x : \alpha \in \Gamma) \)
  - \( \text{exp} \) is a program expression
  - \( \vdash \) is a type to be assigned to \( \text{exp} \)
  - \( \vdash \) pronounced “turnstile,” or “entails” (or “satisfies” or, informally, “shows”)
### Axioms - Constants

\[ \Gamma |- n : \text{int} \]  
(assuming \( n \) is an integer constant)

\[ \Gamma |- \text{true} : \text{bool} \quad \Gamma |- \text{false} : \text{bool} \]

- 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

**Primitive operators** (\( \oplus \in \{ +, -, *, \ldots \} \)):  
\[ \Gamma |- e_1 : \tau_1 \quad \Gamma |- e_2 : \tau_2 \quad (\oplus) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \]

\[ \Gamma |- e_1 \oplus e_2 : \tau_3 \]

**Relations** (\( \sim \in \{ <, >, =, <=, >= \} \)):  
\[ \Gamma |- e_1 : \tau \quad \Gamma |- e_2 : \tau \]

\[ \Gamma |- e_1 \sim e_2 : \text{bool} \]

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

---

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

What do we need to show first?

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

---

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

How to finish?

\( \{x : \text{int}\} |- x : \text{int} \quad \{x : \text{int}\} |- 2 : \text{int} \)

\( \{x : \text{int}\} |- x + 2 : \text{int} \quad \{x : \text{int}\} |- 3 : \text{int} \)

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

Complete Proof (type derivation)

\begin{align*}
\text{Var} & \quad \text{Const} \\
\{x:int\} |- x : int & \quad \{x:int\} |- 2 : int \\
\{x : int\} |- x + 2 : int & \quad \{x:int\} |- 3 : int \\
\{x:int\} |- x + 2 = 3 : bool
\end{align*}

Simple Rules - Booleans

Connectives

\[
\begin{align*}
\Gamma |- e_1 : \text{bool} & \quad \Gamma |- e_2 : \text{bool} \\
\Gamma |- e_1 \&\& e_2 : \text{bool} \\
\Gamma |- e_1 || e_2 : \text{bool}
\end{align*}
\]

Type Variables in Rules

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

Function Application

- Application rule:
  \[
  \Gamma |- e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma |- e_2 : \tau_1 \\
  \Gamma |- (e_1 \ e_2) : \tau_2
  \]
- If you have a function expression \(e_1\) of type \(\tau_1 \rightarrow \tau_2\) applied to an argument \(e_2\) of type \(\tau_1\), 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:
  \[
  \{x : \tau_1\} + \Gamma |- e : \tau_2 \\
  \Gamma |- \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2
  \]

Fun Examples

\[
\begin{align*}
\{y : \text{int}\} + \Gamma |- y + 3 : \text{int} \\
\Gamma |- \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}
\end{align*}
\]
\[
\{f : \text{int} \rightarrow \text{bool}\} + \Gamma |- f 2 :: \text{[true]} : \text{bool list} \\
\Gamma |- (\text{fun } f \rightarrow f 2 :: \text{[true]}) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}
\]
(Monomorphic) Let and Let Rec

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

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

Example

- Which rule do we apply?
  \[ |- (\text{let rec one = 1 :: one in} \quad \text{let } x = 2 \text{ in} \quad \text{fun } y \rightarrow (x :: y :: \text{one}) ) : \text{int} \rightarrow \text{int list} \]

Proof of 1

- Which rule?
  \[ |- (\text{let rec one = 1 :: one in} \quad \text{let } x = 2 \text{ in} \quad \text{fun } y \rightarrow (x :: y :: \text{one}) ) : \text{int} \rightarrow \text{int list} \]

Proof of 3

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

Proof of 3

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

Proof of 3

- Constants Rule
  \[ |- (\text{let rec one = 1 :: one in} \quad \text{let } x = 2 \text{ in} \quad \text{fun } y \rightarrow (x :: y :: \text{one}) ) : \text{int} \rightarrow \text{int list} \]
Proof of 4

Rule for variables

\{one : int list\} |- one : int list

Proof of 2

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

\{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

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

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

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

Proof of 6

Constant

\{\ldots\} |- (::)

: int -> int list -> int list

Variable

\{\ldots; x:int;\ldots\} |- x:int

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

: int list -> int list
Proof of 7

Pf of 6 [y/x]

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

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

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 (\(\tau\)):
  - Basic Types: int, bool, float, string, unit, ...
  - Type Variables: \(\alpha, \beta, \gamma, \delta, \epsilon\)
  - Compound Types: \(\alpha \rightarrow \beta, \text{int} * \text{string}, \text{bool list}, \ldots\)
- Polymorphic Types:
  - Monomorphic types \(\tau\)
  - Universally quantified monomorphic types
  - \(\forall \alpha_1, \ldots, \alpha_n . \tau\)
  - Can think of \(\tau\) as same as \(\forall . \tau\)

Example FreeVars Calculations

- \(\text{Vars('a -> (int -> 'b) -> 'a) = \{\text{a}, \text{'b}\}}\)
- \(\text{FreeVars (All 'b. 'a -> (int -> 'b) -> 'a) = \{\text{a}, \text{'b}\} \text{ \& \& } \{\text{b}\} = \{\text{b}\}}\)
- \(\text{FreeVars \{x : All 'b. 'a -> (int -> 'b) -> 'a, id: All 'c. 'c -> 'c, y: All 'c. 'a -> 'b -> 'c\} = \{\text{a}\} \cup \{\text{a, 'b}\} = \{\text{a, 'b}\} \cup \{\text{b}\}}\)
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
  - Write $\text{FreeVars}(\tau)$
- Free variables of polymorphic type removes variables that are universally quantified
  - $\text{FreeVars}(\forall_1, \ldots, \forall_n. \tau) = \text{FreeVars}(\tau) - \{\forall_1, \ldots, \forall_n\}$
- $\text{FreeVars}(\Gamma) = \text{all FreeVars of types in range of } \Gamma$

10/15/19 49

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 $\Gamma$
  - $\text{Gen}(\tau, \Gamma) = \forall_1, \ldots, \forall_n. \tau$ where
    - $\{\forall_1, \ldots, \forall_n\} = \text{freeVars}(\tau) - \text{freeVars}(\Gamma)$

10/15/19 50

Polymorphic Typing Rules

- A 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

10/15/19 51

Polymorphic Let and Let Rec

- let rule:
  - $\Gamma |- e_1 : \tau_1 \{x : \text{Gen}(\tau_1, \Gamma)\} + \Gamma |- e_2 : \tau_2$
  - $\Gamma |- (\text{let } x = e_1 \text{ in } e_2) : \tau_2$
- let rec rule:
  - $\{x : \tau_1\} + \Gamma |- e_1 : \tau_1 \{x : \text{Gen}(\tau_1, \Gamma)\} + \Gamma |- e_2 : \tau_2$
  - $\Gamma |- (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2$

10/15/19 52

Polymorphic Variables (Identifiers)

- Variable axiom:
  - $\Gamma |- x : q(\tau)$ if $\Gamma(x) = \forall_1, \ldots, \forall_n. \tau$
- Where $q$ replaces all occurrences of $\forall_1, \ldots, \forall_n$ by monotypes $\tau_1, \ldots, \tau_n$
- Note: Monomorphic rule special case:
  - $\Gamma |- x : \tau$ if $\Gamma(x) = \tau$
- Constants treated same way

10/15/19 53

Fun Rule Stays the Same

- fun rule:
  - $\{x : \tau_1\} + \Gamma |- e : \tau_2$
  - $\Gamma |- \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2$
- Types $\tau_1, \tau_2$ monomorphic
- Function argument must always be used at same type in function body

10/15/19 54
Polymorphic Example

Assume additional constants:
- hd : \(\forall \alpha. \alpha \text{list} \rightarrow \alpha\)
- tl: \(\forall \alpha. \alpha \text{list} \rightarrow \alpha \text{list}\)
- is_empty : \(\forall \alpha. \alpha \text{list} \rightarrow \text{bool}\)
- (::) : \(\forall \alpha. \alpha \rightarrow \alpha \text{list} \rightarrow \alpha \text{list}\)
- [] : \(\forall \alpha. \alpha \text{list}\)

Show:

\[
\begin{aligned}
\{} & \vdash \text{let rec } length = \\
& \quad \text{fun } l \rightarrow \text{if } \text{is_empty } l \text{ then } 0 \\
& \quad \text{else } 1 + \text{length } (\text{tl } l) \\
& \quad \text{in } \text{length } ((::) 2 []) + \text{length } ((::) \text{true } []) : \text{int}
\end{aligned}
\]

Let \(\Gamma = \{\text{length:} \alpha \text{list} \rightarrow \text{int}, \ l: \alpha \text{list}\}\)

Show:

\[
\begin{aligned}
\{} & \vdash \text{if } \text{is_empty } l \text{ then } 0 \\
& \quad \text{else } 1 + \text{length } (\text{tl } l) \\
& \quad \text{in } \text{length } ((::) 2 []) + \text{length } ((::) \text{true } []) : \text{int}
\end{aligned}
\]
Polymorphic Example (3): IfThenElse

Let \( \Gamma = \{ \text{length:} \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

\[
\begin{align*}
(4) & \quad (5) & \quad (6) \\
\Gamma|- \text{is_empty} \ l & \quad \Gamma|- 0: \text{int} & \quad \Gamma|- 1 + \\
& \quad : \text{bool} & \quad \text{length} (\text{tl} \ l) : \text{int} \\
\hline
\Gamma|- \text{if is_empty} \ l \text{ then } 0 & \quad \text{else } 1 + \text{length} (\text{tl} \ l) : \text{int}
\end{align*}
\]

10/15/19 61

Polymorphic Example (4)

Let \( \Gamma = \{ \text{length:} \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

\[
\begin{align*}
? & \quad ? \\
\Gamma|- \text{is_empty} : \alpha \text{ list } \rightarrow \text{bool} & \quad \Gamma|- l : \alpha \text{ list} \\
\hline
\Gamma|- \text{is_empty} \ l : \text{bool}
\end{align*}
\]

10/15/19 62

Polymorphic Example (4): Application

Let \( \Gamma = \{ \text{length:} \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

By Const since \( \alpha \text{ list } \rightarrow \text{bool} \) is instance of \( \forall \alpha. \ \alpha \text{ list } \rightarrow \text{bool} \)

\[
\begin{align*}
\Gamma|- \text{is_empty} : \alpha \text{ list } \rightarrow \text{bool} & \quad \Gamma|- l : \alpha \text{ list} \\
\hline
\Gamma|- \text{is_empty} \ l : \text{bool}
\end{align*}
\]

10/15/19 63

Polymorphic Example (4)

Let \( \Gamma = \{ \text{length:} \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

By Const since \( \alpha \text{ list } \rightarrow \text{bool} \) is instance of \( \forall \alpha. \ \alpha \text{ list } \rightarrow \text{bool} \)

\[
\begin{align*}
\Gamma|- \text{is_empty} : \alpha \text{ list } \rightarrow \text{bool} & \quad \Gamma|- l : \alpha \text{ list} \\
\hline
\Gamma|- \text{is_empty} \ l : \text{bool}
\end{align*}
\]

10/15/19 64

Polymorphic Example (5): Const

Let \( \Gamma = \{ \text{length:} \alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list} \} \)

Show

By Const Rule

\[
\begin{align*}
\Gamma|- 0: \text{int}
\end{align*}
\]

10/15/19 66
Polymorphic Example (6): Arith Op

- Let $\Gamma = \{\text{length:}\alpha \text{ list } \rightarrow \text{int}, \ l: \alpha \text{ list}\}$
- Show
  
  By Variable
  
  $\Gamma |- \text{length} \quad (7)$

  By Const
  
  $\alpha \text{ list } \rightarrow \text{int} \quad \Gamma |- (\text{tl} \ l) : \alpha \text{ list}$

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

  $\Gamma |- 1 + \text{length} (\text{tl} \ l) : \text{int}$
Polymorphic Example: (11)AppRule

Let \( \Gamma' = \{ \text{length: } \alpha. \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

By Const since \( \text{int } \rightarrow \text{int list } \rightarrow \text{int list} \) is instance of \( \forall \alpha. \alpha \rightarrow \alpha \text{ list } \rightarrow \alpha \text{ list} \)

\[ \Gamma' |- \lambda : \text{int } \rightarrow \text{int list } \rightarrow \text{int list} \]
\[ \Gamma' |- \lambda : \text{int list } \rightarrow \text{int list} \]

10/15/19

Polymorphic Example: (9)AppRule

Let \( \Gamma' = \{ \text{length: } \alpha. \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

By Var since \( \text{bool list } \rightarrow \text{int} \) is instance of \( \forall \alpha. \alpha \rightarrow \alpha \text{ list } \rightarrow \alpha \text{ list} \)

\[ \Gamma' |- \lambda : \text{bool list } \rightarrow \text{bool list } \rightarrow \text{bool list} \]
\[ \Gamma' |- \lambda : \text{bool list } \rightarrow \text{bool list} \]

10/15/19

Polymorphic Example: (12)AppRule

Let \( \Gamma' = \{ \text{length: } \alpha. \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

By Const since \( \text{bool list } \rightarrow \text{int} \) is instance of \( \forall \alpha. \alpha \rightarrow \alpha \text{ list } \rightarrow \alpha \text{ list} \)

\[ \Gamma' |- \lambda : \text{bool list } \rightarrow \text{bool list } \rightarrow \text{bool list} \]
\[ \Gamma' |- \lambda : \text{bool list } \rightarrow \text{bool list} \]

10/15/19

Polymorphic Example: (13)AppRule

Let \( \Gamma' = \{ \text{length: } \alpha. \alpha \text{ list } \rightarrow \text{int} \} \)

Show:

By Const since \( \text{bool list } \rightarrow \text{bool list} \rightarrow \text{bool list} \) is instance of \( \forall \alpha. \alpha \rightarrow \alpha \text{ list } \rightarrow \alpha \text{ list} \)

\[ \Gamma' |- \lambda : \text{bool list } \rightarrow \text{bool list } \rightarrow \text{bool list} \]
\[ \Gamma' |- \lambda : \text{bool list } \rightarrow \text{bool list} \]

10/15/19