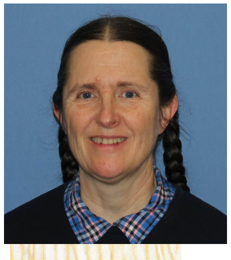
Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC



https://courses.engr.illinois.edu/cs421/sp2023

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,…,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 Inference

- *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

- *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 (Monomorphic)

$$\frac{}{\Gamma \mid\!- \; n : \text{int}}$$ (assuming *n* is an integer constant)

$$\frac{}{\Gamma \mid\!- \; \text{true} : \text{bool}} \qquad \frac{}{\Gamma \mid\!- \; \text{false} : \text{bool}}$$

- 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 (Mono)

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

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

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?

$$\frac{\{x : int\} \mid\!- x + 2 : int \qquad\qquad \{x{:}int\} \mid\!- 3 : int}{\{x{:}int\} \mid\!- x + 2 = 3 : bool}\,\text{Bin}$$

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

How to finish?

$$\dfrac{\dfrac{\{x:int\} \text{ |- } x:int \qquad \{x:int\} \text{ |- } 2:int}{\{x : int\} \text{ |- } x + 2 : int}\,\text{Bin} \qquad \{x:int\} \text{ |- } 3 :int}{\{x:int\} \text{ |- } x + 2 = 3 : bool}\,\text{Bin}$$

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

Complete Proof (type derivation)

$$
\dfrac{\dfrac{\dfrac{}{\{x:int\} \mid\!- x:int}\,\text{Var} \quad \dfrac{}{\{x:int\} \mid\!- 2:int}\,\text{Const}}{\{x : int\} \mid\!- x + 2 : int}\,\text{Bin} \qquad \dfrac{}{\{x:int\} \mid\!- 3 :int}\,\text{Const}}{\{x:int\} \mid\!- x + 2 = 3 : bool}\,\text{Bin}
$$

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;\&\&\; e_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \qquad \Gamma \mid\!- \; e_2 : \text{bool}}{\Gamma \mid\!- \; e_1 \;||\; e_2 : \text{bool}}$$

Type Variables in Rules

- If_then_else rule:

$$\frac{\Gamma \;|\!\!-\; e_1 : \text{bool} \quad \Gamma \;|\!\!-\; e_2 : \tau \quad \Gamma \;|\!\!-\; e_3 : \tau}{\Gamma \;|\!\!-\; (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- 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

Example derivation: if-then-else-

- Γ = {x:int, int_of_float:float -> int, y:float}

$$\frac{\Gamma \mid\!- \text{(fun y -> y > 3) x : bool} \qquad \Gamma \mid\!- \text{x+2 : int} \qquad \Gamma \mid\!- \text{int\_of\_float y : int}}{\Gamma \mid\!- \text{if (fun y -> y > 3) x then x + 2 else int\_of\_float y : int}}$$

Function Application

- Application rule:

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

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

Example: Application

- Γ = {x:int, int_of_float:float -> int, y:float}

$$
\frac{\Gamma \mid\!\!- \text{(fun y -> y > 3)} : \text{int -> bool} \qquad \Gamma \mid\!\!- \text{x : int}}{\Gamma \mid\!\!- \text{(fun y -> y > 3) x : bool}}
$$

Fun Rule

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

$$\frac{\{x : \tau_1\} + \Gamma \ |\!\!- \ e : \tau_2}{\Gamma \ |\!\!- \ \text{fun } x \text{ -> } e : \tau_1 \to \tau_2}$$

Fun Examples

$$\frac{\{y : int\} + \Gamma \text{ |- } y + 3 : int}{\Gamma \text{ |- fun } y \text{ -> } y + 3 : int \rightarrow int}$$

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

(Monomorphic) Let and Let Rec

- let rule:

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

- let rec rule:

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

Example

- Which rule do we apply?

$$\frac{?}{\{\} \mid\!- (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \to (x :: y :: \text{one})) : \text{int} \to \text{int list}}$$

Example

- Let rec rule:

2 {one : int list} |-
(let x = 2 in
fun y -> (x :: y :: one))
: int $\rightarrow$ int list

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

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

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

Proof of 1

- Which rule?

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

Proof of 1

- Binary Operator

3

$$\dfrac{\begin{array}{cc} \{one : int\ list\} \vdash & \{one : int\ list\} \vdash \\ 1 : int & one : int\ list \end{array}}{\{one : int\ list\} \vdash (1 :: one) : int\ list}$$

4

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

Proof of 1

$$\dfrac{\dfrac{③\quad \text{Constant Rule}}{\{\text{one : int list}\} \mid\!- \; 1\text{: int}} \qquad \dfrac{④\quad \text{Variable Rule}}{\{\text{one : int list}\} \mid\!- \; \text{one : int list}}}{\{\text{one : int list}\} \mid\!- \; (1 :: \text{one}) : \text{int list}}$$

Proof of 2

- Let Rule

$$\frac{\{one : int\ list\} \mid\!- 2:int \qquad \begin{array}{l}\{x:int;\ one : int\ list\} \mid\!- \\ fun\ y\ ->\\ \quad (x :: y :: one)) \\ : int \rightarrow int\ list\end{array}}{\begin{array}{l}\{one : int\ list\} \mid\!- (let\ x = 2\ in \\ \quad fun\ y\ -> (x :: y :: one)) : int \rightarrow int\ list\end{array}}$$

Proof of 2

- Constant

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

{one : int list} |- 2:int

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

Proof of 5

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

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 $\rightarrow$ int list

By the Fun Rule

Proof of 5

6 7

$$\frac{?}{\{y:int;\ x:int;\ one:int\ list\}\ |\!-\ x:int} \qquad \frac{?}{\{y:int;\ x:int;\ one:int\ list\}\ |\!-\ (y :: one) : int\ list}$$

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

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

Proof of 6

6

7

Variable 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

$$\dfrac{\dfrac{?}{\{y{:}int;\ \ldots\}\ |\text{-}\ y{:}int} \qquad \dfrac{\dfrac{?}{\{\ldots;\ one{:}int\ list;\ldots\}}}{|\text{-}\ one : int\ list}}{\{y{:}int;\ x{:}int;\ one : int\ list\}|\text{-}\ (y :: one) : int\ list}$$

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

Proof of 7

Variable Rule

{...; one:int list;...}

|- one : int list

Variable Rule

{y:int; ...} |- y:int

{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

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

- Application

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