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 \( \{ x | 2^{15} - 1 \geq x \geq -2^{15} \} \)
  - 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
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, OCAMLR “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 \texttt{op(arg1,\ldots,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
  - 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
- \( \tau \) is a type to be assigned to \( \text{exp} \)
- \( |- \) pronounced “turnstyle”, or “entails” (or “satisfies” or, informally, “shows”)

9/27/22
Axioms – Constants (Monomorphic)

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

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

- 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:

$$\Gamma \vdash x : \sigma \quad \text{if} \quad \Gamma(x) = \sigma$$
Simple Rules – Arithmetic (Mono)

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

\[
\begin{array}{c}
\Gamma |- e_1 : \tau_1 \\
\Gamma |- e_2 : \tau_2 \\
\Gamma |- (\oplus) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3
\end{array}
\]

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

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

\[
\begin{array}{c}
\Gamma |- e_1 : \tau \\
\Gamma |- e_2 : \tau \\
(\sim) : \tau \rightarrow \tau \rightarrow \text{bool}
\end{array}
\]

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

For the moment, think \(\tau\) is \text{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: $\{x : \text{int}\} |- x + 2 = 3 : \text{bool}$

What do we need for the left side?

$\{x : \text{int}\} |- x + 2 : \text{int}$  $\{x: \text{int}\} |- 3 : \text{int}$

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

How to finish?

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

Complete Proof (type derivation)

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

Connectives

\[
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \\
\Gamma \vdash e_1 \land e_2 : \text{bool}
\]

\[
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \\
\Gamma \vdash e_1 \lor e_2 : \text{bool}
\]
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:

\[
\Gamma |- e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma |- e_2 : \tau_1 \\
\hline
\Gamma |- (e_1 e_2) : \tau_2
\]

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:

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

\[
\begin{align*}
\Gamma |- e_1 : \tau_1 & \quad \{ x : \tau_1 \} + \Gamma |- e_2 : \tau_2 \\
\Gamma |- (\text{let } x = e_1 \text{ in } e_2) : \tau_2
\end{align*}
\]

- let rec rule:

\[
\begin{align*}
\{ x : \tau_1 \} + \Gamma |- e_1 : \tau_1 & \quad \{ x : \tau_1 \} + \Gamma |- e_2 : \tau_2 \\
\Gamma |- (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2
\end{align*}
\]