Types in Programming Languages
Types in Programming Languages

Organization of **untyped values**
- At the lowest level, everything is a sequence of bits
- Need higher-level view: categorize these bit sequences based on usage and behavior

Type = set of values with uniform behavior

**Type-related constraints** to enforce correctness, e.g:
- Should not try to multiply two strings
- Should not use a character value as a condition of an if-statement
- Should not use an integer as a pointer
Static Typing

Statically typed languages: expressions in the code have **static types**

- static type = promise about run-time values
- Types are either **declared** or **inferred**
- Examples: C, C++, Java, ML, Pascal, Modula-3

A statically typed language typically does some form of **static type checking**

- E.g., at compile time Java checks that the [] operator is applied to a value of type “array”
Dynamic Typing

Dynamically-typed languages: entities in the code do not have static types

- Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python
- Entities in the code do not have declared types, and the compiler does not try to infer types for them

Dynamic type checking

- Before an operation is performed at run time
- E.g., in Scheme: \((+ 5 \#t)\) fails at run time, when the evaluation expects to see two numeric values as operands of +
Examples of Types

Integers

Arrays of integers

Pointers to integers

Records with fields `int x` and `int y`
  e.g., “struct” in C

Objects of class C or a subclass of C
  e.g., C++, Java, C#

Functions from any list to integers
Numeric Types  [no need to remember this]

C does not specify the ranges of numeric types
- Integer types: char, short, int, long, long long
  - Includes “unsigned” versions of these
- Floating-point types: float, double, long double

Java specifies the ranges of numeric types
- byte: 8-bit signed two's complement integer [-128,+127]
- short: 16-bit signed two's complement integer [-32768,+32767]
- int: 32-bit signed two's complement integer [-2147483648,+2147483647]
- long: 64-bit signed two's complement integer [-9223372036854775808, +9223372036854775807]
- float/double: single/double-precision 32-bit IEEE 754 floating point
- char: single 16-bit Unicode character; minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65535)
Enumeration Types [no need to remember this]

C: a set of named integer constant values
   – Example from the C specification
     enum hue { chartreuse, burgundy, claret=20, winedark };
     /* the set of integer constant values is { 0, 1, 20, 21 } */
     enum hue col, *cp;
     col = claret; cp = &col;
     if (*cp != burgundy) ...

Java: a fixed set of named items (not integers)
     enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                 THURSDAY, FRIDAY, SATURDAY }
   – In reality, it is like a class: e.g., it can contain methods
Types as Sets of Values

32-bit Integers: the set of numbers that can be represented in 32 bits in signed two’s-complement

- “type int” = \{ -2^{31}, ..., 2^{31} - 1 \}

Class type: any instance of class C or a subclass of C

- “type C” = set of all instances of C or of any transitive subclass of C [not the same as “class C”, which is just a blueprint for creating objects]

Static type of an expression/variable: “at run time, the expression/variable values will be from this set”

Subtypes are subsets: T2 is a subtype of T1 if T2’s set of values is a subset of T1’s set of values
Monomorphism vs. Polymorphism

Greek: mono = single; poly = many; morph = form

Monomorphism
– Every value belongs to exactly one type

Polymorphism
– A value can belong to multiple types

Typical example: subtype polymorphism
– E.g., class X, class Y extends X, class Z extends Y
– Class type X = all instances of X, Y, and Z
– Class type Y = all instances of Y and Z [subtype of type X]
– An instance of class Z belongs to all three class types
– Variable of static type X is really of type “reference to values of type X”, so it can refer to X, Y, or Z instances
More Polymorphism

Parametric polymorphism
- E.g. f(x)=x has types \( \text{Bool} \rightarrow \text{Bool} \), \( \text{Nat} \rightarrow \text{Nat} \), ...
- Use a type parameter \( T \); define type type \( T \rightarrow T \)
- Generics in C++ and Java – e.g. Map\(<K,V>\)
- ML and similar functional languages

Coercion: values of one type are silently converted
- E.g. addition: 3.0 + 4 : converts 4 to 4.0
- In a context where the type of an expression is not appropriate: (1) either an automatic coercion to another type is performed automatically; (2) or if not possible: compile-time error
Coercions

Widening
– coercing a value into a “larger” type
– e.g., \texttt{int} to \texttt{float}, subtype to supertype

Narrowing
– coercing a value into a “smaller” type
– loses information, e.g., \texttt{float} to \texttt{int}
Widening Primitive Conversions in Java [no need to remember this]

Widening primitive conversions
- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

“integral type to integral type” and “float to double” do not lose any information
Widening Primitive Conversions in Java [no need to remember this]

Language Spec says

– Conversion of an int or long value to float, or of a long value to double, may result in loss of precision
– The result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode
Contexts for Widening Conversions

**Assignment conversion**: when the value of an expression is assigned to a variable

**Method call conversion**: applied to each argument value in a method or constructor invocation
  - The type of the argument expression must be converted to the type of the corresponding formal parameter

**Casting conversion**: applied to the operand of a cast operator: (float) 5
Contexts for Widening Conversions

**Numeric promotion**: converts operands of a numeric operator to a common type

- Example: binary numeric promotion [no need to remember this]
  - e.g. +, -, *, etc.
  - If either operand is double, the other is converted to double
  - Otherwise, if either operand is of type float, the other is converted to float
  - Otherwise, if either operand is of type long, the other is converted to long
  - Otherwise, both are converted to type int
Narrowing Conversions [no need to remember this]

Narrowing primitive conversions in Java
- e.g. long to byte, short, char, or int
- float to byte, short, char, int, or long
- double to byte, short, char, int, long, or float

Examples of loss of information
- int to short: loses high bits
- int not fitting in byte: changes sign and magnitude
- double too small for float: underflows to zero
Type Systems
Type System and Type Checking

Based on the set of types, a type system can prove that programs are “good” without running them. A well-typed program will not “go wrong” at run time.

Works only for some run-time errors, not all:

E.g. cannot assure the absence of “division by zero” or “array index out of bounds” - they depend on particular values from a type.

But can catch type-related errors such as “addition with a boolean operand” error.

In reality, it is a simple form of abstract interpretation.
Simple Language  (from the programming projects)

<expr> ::= const  |  id  [only consider integer vars/consts; in the project also do float]

|  <expr> + <expr>  |  <expr> - <expr>
|  <expr> * <expr>  |  <expr> / <expr>
|  (  <expr>  )

<cond> ::= true  |  false  |  <expr> < <expr>  [also <=, >=, ==, !=]

|  <cond> && <cond>  |  <cond> || <cond>
|  ! <cond>  |  (  <cond>  )
Simple Abstract State

Abstract state: a map $\sigma_a$ from vars to abstract values
A summarization of many possible concrete states

$\sigma_a : \text{Vars} \rightarrow \{ \text{Int}, \text{Float} \}$ These are just $\text{AnyInt}$ and $\text{AnyFloat}$ from earlier

There is only one abstract state, defined by the declarations in the program

State never changes: e.g., if we declare a variable to be of type int, it is of type int everywhere, in all executions

Compare this with the more refined abstract interpretation from earlier, where a variable can have different abstract values at different program points
Abstract Evaluation

Abstract evaluation relation for arithmetic expressions: triples $<\text{ae}, \sigma_a> \rightarrow v_a$

- $\text{ae}$ is a parse subtree derived from $<\text{expr}>$
- $\sigma_a$ is the abstract state defined by declarations
- $v_a$ is an abstract value $\in \{\text{Int}, \text{Float}\}$

Meaning of $<\text{ae}, \sigma_a> \rightarrow v_a$: the evaluation of $\text{ae}$ from any concrete state $\sigma$, if it completes successfully, will produce a concrete value $v$ abstracted by $v_a$

Example: $<x+y+1, [x\mapsto\text{Int}, y\mapsto\text{Int}]> \rightarrow \text{Int}$

Example: $<x*y-1.2, [x\mapsto\text{Float}, y\mapsto\text{Float}]> \rightarrow \text{Float}$
Evaluation for Arithmetic Expressions

Syntax: \( \text{id} \mid \text{const} \mid \text{<expr>} + \text{<expr>} \mid \ldots \)

- \( \text{<const}, \sigma_a \rightarrow \text{Int} \) if const.lexval is an integer constant; similarly for \( \text{Float} \)

- \( \text{id}, \sigma_a \rightarrow \sigma_a(\text{id}) \) static error if \( \sigma_a(\text{id}) \) is undefined; use of undeclared variable

\[
\begin{align*}
\text{<ae}_1, \sigma_a & \rightarrow v_{a1} \\
\text{<ae}_2, \sigma_a & \rightarrow v_{a2} \\
\text{<ae}_1 + \text{ae}_2, \sigma_a & \rightarrow v_a
\end{align*}
\]

\[ v_a = v_{a1} +_a v_{a2} \]

Here we use abstract addition operator \( +_a \) working on abstract values
Evaluation for Expressions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Int</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Int</td>
<td>![Alert]</td>
</tr>
<tr>
<td>Float</td>
<td>![Alert]</td>
<td>Float</td>
</tr>
</tbody>
</table>

Same for arithmetic ops -, *, /

For boolean expressions: introduce abstract value *Bool*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Int</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Bool</td>
<td>![Alert]</td>
</tr>
<tr>
<td>Float</td>
<td>![Alert]</td>
<td>Bool</td>
</tr>
</tbody>
</table>

Same for comparison ops <, >, >=, ==, !=

For boolean ops &&, ||, !: the grammar already makes sure that their operands are of type *Bool*; no need for checking rules
Typed Expressions

As with other static checking: without evaluating an expression, can we guarantee that its evaluation will not produce a run-time error? (static a.k.a. compile-time analysis)

For our simple language

Type \textit{Int} = \text{set of all expressions that are guaranteed to evaluate to an integer value in the concrete operational semantics (regardless of the concrete state \(\sigma\))}

Similarly for \textit{Float}
Typing Relation

Typing relation for arithmetic expressions: binary relation, a simplified version of the evaluation relation.

If \( \langle \text{ae}, \sigma_a \rangle \rightarrow T \) we will write \( \text{ae} : T \) [here \( T \) is \text{Int} or \text{Float}]

Type checking is defined by inference rules for the typing relation:

- \( \text{const} : T \) if \text{const}.lexval is a constant of type \( T \)
- \( \text{id} : T \) if \text{id} is declared of type \( T \)

\[
\begin{align*}
\text{ae}_1 : T & \quad \text{ae}_2 : T \\
\hline
\text{ae}_1 + \text{ae}_2 : T
\end{align*}
\]

If there is a derivation tree for \( \text{ae} : T \), the expression is well-typed and will produce a value of type \( T \) at run time.
Static Type Safety

If a program is well-typed, we can guarantee the absence of certain type-related errors

Static type safety: all bad behaviors of certain type-related kinds are excluded - e.g., Java, but not C

Example: C is not type safe

E.g., \texttt{double pi = 3.14; int* ptr = (int*) \&pi; int x = *ptr;}

This program will be type checked successfully – but typecasting “pointer to float” into “pointer to int” at run time will produce a garbage value in \texttt{x}
Language Safety

Want more than static type safety – want language safety

Cannot “break” the abstractions of the language (type-related and otherwise); e.g. no buffer overflows, segmentation faults, return address overriding, garbage values, etc.

Example: C is unsafe for many reasons, one of which is the lack of static type safety

Other reasons: null pointers lead to segmentation faults (OS concept, not PL concept); buffer overflows lead to stack smashing & garbage values
Language Safety

Example: **Java is safe** – combination of static type safety & run-time checks

Static type safety ensures that an well-typed program will not do type-related “bad” things

Run-time checks catch things that cannot be caught statically via types: e.g., null pointers, array index out of bounds, division by zero

Example: **Lisp is safe** – dynamic checks for type-related correctness (“operands of PLUS must be numbers”) and special “bad” values (e.g. “trying to get an element of an empty list”)